Manual de Programación Lógica

Introducción a la programación con la ayuda de PSeInt

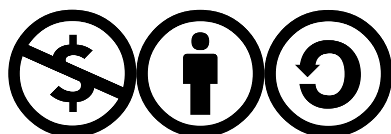
Manual de Programación Lógica

Introducción a la Programación
con la ayuda de PseInt

Lic. Ricardo Saucedo

30 de julio de 2015
Documento en proceso de elaboración.





Índice de contenido

Presentación

El objetivo de esta guía es ayudar a los alumnos de Programación Lógica a la comprensión de la mecánica de resolución de problemas básicos de programación y guiarlos en la tarea de diagramar y codificar los algoritmos mediante pseudocódigo.

Se utilizará como herramienta de estudio el programa PseInt versión de fecha 07/04/2015. El programa está disponible en forma gratuita en el sitio http://pseint.sourceforge.net/ cuyo autor, Pablo Novara (zaskar_84@yahoo.com.ar) lo publica bajo licencia GPL.

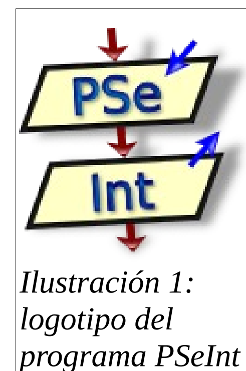


Ilustración 1: logotipo del programa PSeInt

En el sitio web se indica que esta herramienta está diseñada para asistir a un estudiante en sus primeros pasos en programación. Mediante un simple e intuitivo pseudolenguaje en español (complementado con un editor de diagramas de flujo), le permite centrar su atención en los conceptos fundamentales de la algoritmia computacional, minimizando las dificultades propias de un lenguaje y proporcionando un entorno de trabajo con numerosas ayudas y recursos didácticos.

Instalación del programa.

Debemos hacer una descarga del programa en su versión más actualizada a partir de la página del autor, se deberá elegir la descarga correspondiente a nuestro sistema operativo, GNU Linux / Windows / Mac OS. Al finalizar la descarga nos encontraremos con un archivo para instalar en el caso de windows, y para descomprimir en el caso de utilizar linux. De todas formas en la misma página de la descarga hay una explicación muy completa que nos ayudará a la instalación.

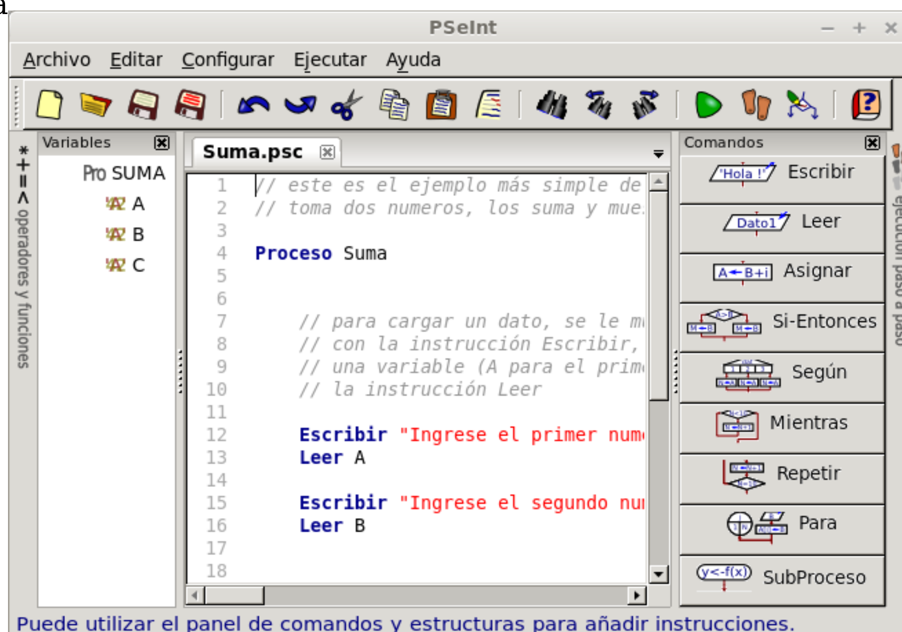


Ilustración 2: Ventana del programa PSeInt

Introducción a la Programación

Se define como **programación** al proceso de dotar a la computadora de un método para resolver un problema tipo y entregar un resultado.

Se dice también que un programa está compuesto de dos partes:

- El **código,** que consiste en el conjunto de acciones a ejecutar, en programación a cada acción básica se la denomina genéricamente como **instrucción**. Cada instrucción podrá estar compuesta de un conjunto de elementos los cuales estudiaremos más adelante.
- La **estructura de datos** que consiste en la disposición de memoria elegida para almacenar los datos necesarios para resolver el problema, ya sea los parámetros o **datos de entrada**, los resultados intermedios o **datos de proceso** y la información resultante o **datos de salida**. A estos datos en programación se los denomina genéricamente como **variables**.

El computador es una máquina que por sí sola no puede hacer nada, necesita ser programada, es decir, introducir a su memoria instrucciones que la guíen en lo que tiene que hacer. Esta es una tarea que para un neófito puede resultar muy compleja ya que la computadora solamente es capaz de realizar tareas muy básicas, por lo tanto hay que definir claramente cual será al proceso a ejecutar para hacer tareas complejas mediante pasos elementales.

Para comprenderlo más organizadamente, podemos decir que este proceso de programación se divide en las siguientes tareas:

1. Comprensión del problema.
2. Idear una solución que sea lo más general posible.
3. Enunciar un algoritmo de resolución.
4. Codificarlo en un lenguaje de programación.
5. Implementarlo, o dicho de otra manera, hacer que la máquina pueda traducirlo a su propio código binario.
6. En forma paralela a los pasos anteriores, preparar un lote de prueba.
7. Ejecutar el programa y validarlo mediante el lote de prueba, si se encuentran errores lógicos, corregirlos, o sea, volver al punto 3 o 4 según el nivel de error.

Comprensión del Problema.

En esta etapa, se debe confeccionar el **enunciado** del problema, el cual requiere una definición clara y precisa. Es importante que se conozca exactamente cual es el objetivo que debe resolverse o dicho de otra manera, lo que se desea que realice la computadora.

Quizás quién enuncie el problema no sepa definir claramente los objetivos, así que el analista deberá interrogarlo hasta que el objetivo a resolver quede perfectamente claro, mientras esto no se logre completamente, no tiene mucho sentido continuar con la siguiente etapa.

Encontrar una solución

Normalmente esta etapa comienza en paralelo a la lectura del enunciado, cuando se va comprendiendo el enunciado y por experiencia o analogía se van ideando las acciones que se deben aplicar para arribar al resultado esperado.

La solución debe ser lo más amplia o general posible, se debe realizar una abstracción del problema, por ejemplo si el enunciado dice:

> *Se necesita un programa para calcular el área ocupada por una viga de madera de 3 metros de largo, y sección cuadrada de 12 cm de lado.*

Nos encontramos ante una problema de tipo particular, para que éste sirva a los efectos de un desarrollo informático habría que generalizarlo. Así el enunciado podría ser transformado en:

> *Se necesita un programa para calcular el área ocupada por una viga de constitución uniforme, dados su largo en metros, y el tamaño en centímetros de cada lado de la sección.*

Es necesario en esta etapa tener en claro:

- Cuales son los datos que se requieren para definir el caso particular, los denominados datos de entrada.
- Cual es el conjunto de resultados que se desea obtener, o sea, la información de salida.
- Los métodos y fórmulas que se necesitan para procesar los datos.

Más adelante entraremos en detalle en las características inherentes a los datos para comprender como esos datos pueden ser acotados y así minimizar la probabilidad de error de proceso.

Enunciar un Algoritmo

Se debe plantear el método de resolución elegido mediante una construcción matemática denominada **algoritmo**.

Un aspecto importante a tener en cuenta en el enunciado de un algoritmo es como se administrarán los datos definidos en el problema. Siempre que un **evento** (un acontecimiento, una verificación, una medición, etc.) se pueda registrar de alguna forma, este evento se considerará como computable. Al registro de un evento en un sistema informático se lo denomina **dato**. Podemos diferenciar en un algoritmo dos tipos de datos, las constantes y los datos variables. Analicemos un ejemplo:

Si queremos hacer un simple algoritmo para calcular el perímetro de una circunferencia lo podríamos definir en los siguientes pasos:

1. Establecer como dato de entrada el *radio* de la circunferencia.
2. Establecer como fórmula de resolución: *circunferencia* = 2 x PI x *radio*
3. Establecer como dato de salida la *circunferencia* calculada.

En este caso tenemos dos datos perfectamente definidos: *radio* y *circunferencia*, ambos son datos **variables** ya que <u>no podemos saber de</u>

ninguna forma el valor exacto del radio, pues este valor va a depender de un evento que el usuario determinó, luego la circunferencia (que depende del valor del radio) tampoco puede ser conocida de antemano. Por eso en el cálculo a los datos se los incorpora en forma *referencial* a través de un nombre simbólico.

Por otra parte en la fórmula observamos dos datos que se requieren para el cálculo, el valor *2* y el valor *PI* (3,1415926...) ambos valores están perfectamente definidos y no es posible que cambien ya que entonces el resultado obtenido sería incorrecto. Ambos valores son entonces **constantes** y se incorporan *literalmente* en el cálculo.

Además para enunciar este algoritmo se utilizará una herramienta de representación llamada **diagrama** o bien mediante un **pseudocódigo**. Veamos esto por partes:

Algoritmo

Un **algoritmo** (del griego y latín, *dixit algorithmus* y este a su vez en honor del matemático persa *Al-Juarismi*) se define como un método aplicable con el fin de obtener la solución a un problema genérico. Debe presentarse como una secuencia ordenada de pasos que siempre se ejecutan en tiempo finito y con una cantidad de esfuerzo también finito o al menos que pueda detectar cuando el método se hace inviable e informarlo. Los algoritmos tienen un inicio y un final, son únicos y deben ser fácilmente identificables.

Para su mejor claridad y aprovechamiento en la etapa de codificación, es conveniente que los algoritmos tengan ciertas características:

- Debe tener un **único comienzo** y un sólo **final** perfectamente definidos.
- Debe ser **secuencial**, cada paso se debe ejecutar en una forma ordenada y no debe comenzar a ejecutarse un paso sin haber concluido de ejecutar el paso anterior. Cuando se hace el diagrama del algoritmo, a esta secuencialidad se la denomina **flujo de proceso**.
- Deben ser **Finitos**, con lo cual se entiende que deben finalizar en algún momento determinado.
- Deben ser **Eficientes**, entendiéndose por esto que ocupen la cantidad mínima y necesaria de variables para que al codificar genere un programa que utilice la mínima cantidad de memoria y además minimizar el tiempo de ejecución evitando procesos redundantes o innecesarios.
- Deben ser **Legibles**, se busca que el texto que lo describe debe ser claro, de forma que permita entenderlo y leerlo fácilmente, eligiendo nombres de variables y de procesos que hagan referencia clara a su función dentro del código.
- Deben ser **Modificables**, o sea que deben enunciarse de modo que sus posteriores modificaciones u ampliaciones sean fáciles de realizar, incluso por programadores diferentes a sus propios autores.
- Deben ser **Modulares**, la filosofía utilizada para su diseño debe favorecer la división del problema en módulos pequeños. haciendo que procesos y cálculos complejos se descompongan en cálculos más simples. Si bien esto puede sonar contrario al enunciado sobre la eficiencia, lo

óptimo sería poder llegar a un compromiso entre simplicidad y eficiencia.

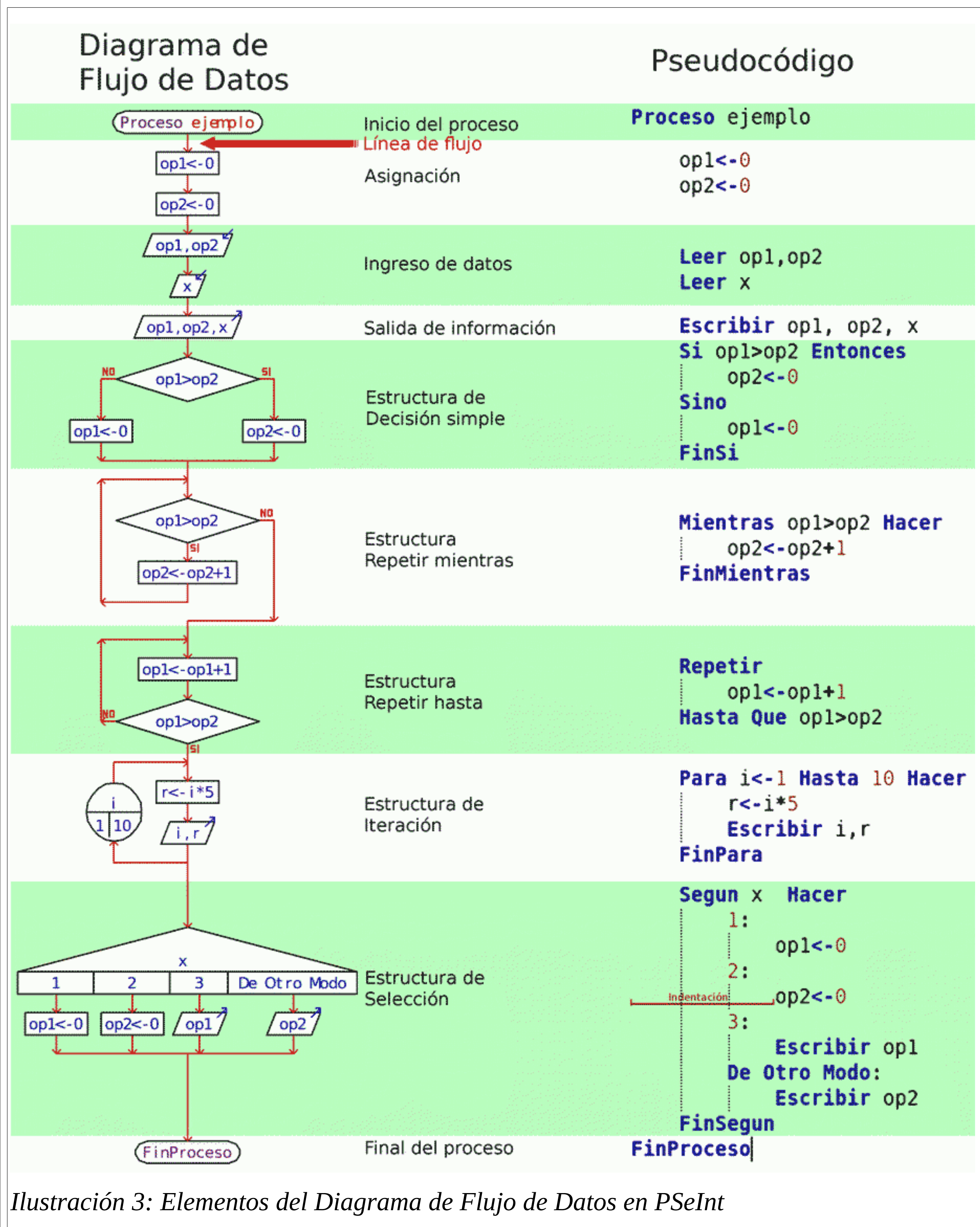


Ilustración 3: Elementos del Diagrama de Flujo de Datos en PSeInt

Características de un Algoritmo computacional

Para poder enunciar un algoritmo que se pueda codificar en un lenguaje de programación, debemos asumir que quién interpretará el algoritmo (la CPU) podrá realizar solamente acciones muy básicas. Si bien en los lenguajes de programación nos encontramos en que podemos utilizar multitud de acciones

predefinidas, la mayoría están asociadas al contexto de ejecución o dicho de otro modo, la mayoría de acciones que se pueden realizar mediante un lenguaje de programación son de “forma” y no de “fondo” en cuanto a la resolución esencial de un problema.

Digamos que nos concentraremos solamente en aquellas acciones que son comunes a cualquier lenguaje de programación y no contemplaremos ninguna acción que sea exclusiva de un lenguaje. Así podemos determinar que las acciones que serán permitidas en un algoritmo son las siguientes:

- Introducir un valor en una variable desde un dispositivo de entrada (el teclado por ejemplo)
- Mostrar en un dispositivo de salida un valor ya sea de una variable o un valor literal (en la pantalla, por ejemplo)
- Almacenar un valor en una variable como resultado de una expresión (como se denomina una cálculo en el lenguaje de la programación)
- Bifurcar el flujo de ejecución entre dos alternativas, mediante una condición evaluada.
- Repetir parte del flujo normal de proceso de una manera condicional.
- Iterar, o sea repetir parte del flujo un número determinado de veces.
- Seleccionar un único flujo de proceso a ejecutar de un conjunto de flujos posibles.

Diagrama

Según la Wikipedia, un **diagrama** es una representación gráfica de una proposición, de la resolución de un problema, de las relaciones entre las diferentes partes o elementos de un conjunto o sistema, o de la regularidad en la variación de un fenómeno que permite establecer algún tipo de ley.

Existen muchísimos tipos de diagramas cada uno es aplicable de acuerdo a la información que se quiera representar, así por ejemplo en teoría de conjuntos tenemos los diagramas de Venn, en bases de datos se utilizan los diagramas de Entidad-Relación. En nuestro caso, la programación, se utilizan principalmente dos tipos de diagramas, los diagramas de Flujo de Datos y los de Nassi-Shneiderman, estos últimos los dejaremos fuera de nuestro análisis por ahora y nos concentraremos exclusivamente en los diagramas de flujo.

DFD - Diagramas de Flujo de Datos

Si bien los diagramas de flujo se utilizaban mucho tiempo antes de que la informática se volviera una realidad, es gracias a Herman Goldstine y John von Neumann la aplicación de estos diagramas a la planificación de algoritmos computacionales. Precisamente por su aplicación a la informática y su estandarización, en la actualidad los diagramas de flujo se utilizan en muchos campos del conocimiento fuera de la programación, por ejemplo, en el ámbito empresarial, en medicina, etc.

En los diagramas de flujo de datos (abreviado DFD), el flujo precisamente está indicado por una línea descendente que indica la secuencia de ejecución de las instrucciones.

En la ilustración 3 podemos observar todos los elementos del diagrama de flujo que maneja el programa PseInt. En el siguiente capítulo iremos analizando con ejemplos la funcionalidad de cada elemento del diagrama.

Pseudocódigo

El **Pseudocódigo** es otra forma de representar un algoritmo, en este caso utilizando una notación similar a un lenguaje de programación estándar, pero cuyas palabras claves están escritas en el lenguaje castellano. Existe una correspondencia entre los elementos del DFD y el Pseudocódigo como se puede apreciar en la ilustración 3.

La idea es que se pueda ver el programa desde el punto de vista del programador el cual está acostumbrado a leer **código fuente**.

Estructura de un programa

Habíamos mencionado previamente que en un programa encontramos dos partes, el código y la estructura de datos.

El código

El **código** podemos verlo como un texto que describe la receta que se debe aplicar para realizar una tarea compleja de procesamiento de datos. Pero... ¿Con qué objetivo se define este texto?

La idea es que la computadora puede ejecutar programas que traducen texto generado por el usuario (denominado **código fuente**) a operaciones en **código máquina** (denominado **código objeto**) y puede ejecutar cada acción descripta en la cada oración. Estos programas traductores a código máquina se denominan genéricamente como programas **intérpretes**. El único problema es que los lenguajes naturales son poco precisos al momento de definir tareas, ya que nos encontramos con que podemos utilizar un conjunto de sinónimos para representar lo mismo o incluso una misma palabra presenta ambivalencias, es decir en un contexto significa una cosa y en otro contexto otra cosa. Para no hacer demasiado complejo al programa intérprete, estos reconocen un lenguaje muy acotado (que se basa en su amplia mayoría en el idioma inglés) resumiendo el vocabulario normal a unas pocas palabras con un significado preciso, incluso muchas de esas palabras son inexistentes en el lenguaje natural y solo tienen sentido en la programación. El objetivo es hacer que el texto que podamos armar no sea difícil de entender y traducir por no tener ninguna ambivalencia en su significado. Como todo texto, un programa no es más que un conjunto de oraciones, en este caso un programa se compone de oraciones de tipo declarativas e imperativas.

Retomando la idea entonces, cuando programamos no hacemos más que describir paso por paso y con oraciones muy simples que debe hacer o tener en consideración la computadora para aplicar un algoritmo. En nuestra jerga vamos a denominar a la oraciones declarativas como **sentencias** y a las oraciones imperativas como **instrucción**.

Como todo lenguaje, la oración está sujeta a leyes sintácticas que en este caso son muy estrictas. Por eso la primer tarea que el programa traductor hace, luego de que nosotros codifiquemos el programa es realizar un análisis sintáctico de todo el texto o de cada oración indicando todos los errores sintácticos detectados para que los corrijamos. Solo cuando el programa se haya libre de errores sintácticos se puede comenzar con la traducción a código máquina y posterior ejecución.

Sintaxis

Las reglas sintácticas son mas bien pocas, pero muy restrictivas. Cada lenguaje tiene sus propias restricciones. En el caso del pseudocódigo podemos mencionar las siguientes reglas:

Se define un conjunto de **palabras reservadas** que en algunos lenguajes son

denominadas **tokens**. Estas palabras solo pueden utilizarse en un único sentido, el que se asigne como definición en el lenguaje.

Las palabras reservadas y su significado son:

Palabra reservada	*Significado*
Leer *<v>*	Ingresar un conjunto de datos desde el teclado hacia variables de memoria, se menciona al teclado como generalización de cualquier periférico de entrada.
Escribir *<e>*	Exhibir por la pantalla de video un conjunto de datos, se indica al video como una generalización de dispositivo de salida.
<v> **<-** *<e>*	Asignación de un dato a una variable de memoria
Si *<e>* **Entonces** **SiNo** **FinSi**	Bifurcar el flujo normal de operación entre dos flujos alternativos.
Mientras *<e>* **Hacer** **finMientras**	Repetir una secuencia de instrucciones mientras se cumpla una condición
Repetir **Hasta que** *<e>*	Repetir una secuencia de instrucciones hasta que se cumpla una condición
Para *<v>* **<-** *<e>* **Hasta** *<e>* **Hacer** **finPara**	Repetir una secuencia de instrucciones una cantidad fija de veces.
Según *<v>* **hacer** <k_0>: <k_1>: ... <k_n>: **De Otro Modo:** **finSegún**	Bifurcar el flujo normal de operación entre varios flujos alternativos.
Proceso *<n>* **finProceso**	Delimitador de puntos de entrada y de salida de un proceso cuyo nombre es indicado por <n>.

Como observamos en la tabla de palabras reservadas, utilizamos unos símbolos para indicar elementos obligatorios, vamos a definirlos también:

<v> indica que se debe especificar un dato variable (o una variable a secas) el cual representa un espacio en la memoria del equipo donde se almacenará un valor.

<k> indica que se deberá utilizar un valor constante, o sea que tiene que estar definido literalmente.

<e> indica una **expresión**, o sea, el resultado de un cálculo entre uno o varios datos. Una expresión además de un resultado también puede ser un valor constante o variable.

<n> palabra definida por el usuario (p.ej. un nombre de variable o de proceso).

Otros elementos que forman parte del código son los **operadores**, los **delimitadores** y las **funciones**. Más adelante hablaremos de ellos.

Instrucciones

Son oraciones imperativas cuya estructura es siempre: "hacer tal cosa con tales datos". El núcleo de la oración es el verbo que indica la orden que queremos que se ejecute, lo llamaremos precisamente **orden** o **comando**. Los

verbos tienen un solo modo, el infinitivo.

En los lenguajes de programación hay multitud de comandos pero en el pseudocódigo vamos a enfocarnos solamente en tres ya mencionados:

- Leer
- Escribir
- <- (asignar)

Sentencias

Son oraciones declarativas que sirven para dividir la secuencia de ejecución en bloques que incluyan instrucciones u otras sentencias (a estas se las denomina sentencias anidadas) que se ejecutaran en forma grupal, delimitando donde comienza y donde termina este bloque. Por comodidad denominaremos a este conjunto como **estructura**. En este caso, la existencia de estas sentencias definen a la metodología de programación como **programación estructurada**.

Las sentencias son

```
• Proceso … / finProceso
• Si … Entonces / Sino /finSi
• Mientras ... Hacer / finMientras
• Repetir / Hasta que ...
• Para … <- … Hasta ... Hacer / finPara
• Según … Hacer / deOtroModo: / finSegún
```

Estructura de datos

La estructura de datos está formada por el conjunto de datos **variables** que requiere el programa. Los datos constantes no forman parte de la estructura de datos ya que estos, por el mismo hecho de ser constantes, se almacenan como parte del código.

Los lenguajes de programación definen su propias reglas para la definición de las estructuras de datos, pero en el caso del pseudocódigo resumiremos esas reglas en la siguientes:

Las variables se definen por un **nombre** y un **tipo primitivo**.

Respecto al **nombre,** cada variable debe poseer un nombre único, éste debe comenzar por una letra (vocal o consonante) y luego puede continuar con un conjunto de letras o números. Queda prohibido como parte del nombre cualquier símbolo y tampoco se admiten espacios.

Respecto a los **tipos primitivos** de datos, existen tan solo tres:

- El tipo **lógico** o **booleano**, el cual representa un dato que puede tener sólo dos posibles valores: **verdadero** o **falso**.
- El tipo **numérico**, el cual representa un dato que puede operarse matemáticamente, por ejemplo una medida, el resultado de un cálculo, etc. este valor debe pertenece al conjunto de números reales.
- El tipo **alfanumérico**, el cual representa un dato que no es matemático, por ejemplo un nombre, una denominación, un código, etc. a los

alfanuméricos también se los conoce como **strings** o cadenas de texto.

Hay otras características que definen a ciertos tipos de variables como por ejemplo la dimensión, pero eso será objetivo de otro capítulo.

Expresiones

Ya habíamos mencionado que una expresión consiste en el resultado de un cálculo, pero debemos ser un poco más extensos en su definición ya que involucra otros elementos de la programación que no hemos mencionado y también por su importancia ya que es una de la construcciones más utilizadas en los programas.

Una expresión esta compuesta por datos afectados por **operadores** y/o **funciones**.

Un operador es un símbolo que representa una operación entre datos. Algunos son muy conocidos como por ejemplo + y - (la suma y la resta), pero hay muchos más.

Una función es un proceso definido que se hace con uno o varios datos y del cual se obtiene un único resultado. Por ejemplo las funciones trigonométricas seno(), coseno(), etc.

Operadores

Los operadores pueden ser de los siguientes tipos: matemáticos, relacionales, lógicos y por último un operador llamado concatenación.

- **Operadores matemáticos** son la suma (+), la resta (-), la división (/), la multiplicación (*), la potenciación (^) y el módulo o resto de la división (%). El resultado de una operación matemática es un tipo numérico.
- **Operadores relacionales** son: el mayor (>), menor (<) e igual (=) y sus operaciones complementarias: menor o igual (<=), mayor o igual (>=) y distinto (<>) respectivamente. El resultado es un tipo lógico ya que nos dice si la relación es verdadera o falsa.
- **Operadores lógicos**: se utilizan para unir subexpresiones de tipo lógico, ellas son la conjunción AND (&) la disyunción OR (|) y el complemento NOT (~), el resultado es un tipo lógico siguiendo las reglas de las tablas de verdad del álgebra de boole.
- **Operador de concatenación**: es el único operador que permite operar entre si dos datos de tipo alfanumérico, el resultado es una cadena de texto que contiene las dos cadenas unidas una continuación de la otra.

Casi todas las operaciones requieren que existan dos operandos, uno a derecha y otro a izquierda, por eso es que a la mayoría se los clasifica cómo de tipo **binario.** Con las siguientes excepciones :

El operador NOT (~) solo admite un operando, por lo tanto se lo clasifica cómo operador **monario**.

El operador menos (-) se puede utilizar para realizar una sustracción, con lo cual es del tipo binario, pero también se lo puede utilizar como operador de

cambio de signo, actuando en este caso como monario.

Ejemplos:

`A ← b – 4` *(utilizado en forma binaria)* `A ← -b` *(utilizado en forma monaria)*

Una expresión cuando es muy compleja puede analizarse dividiéndola en varias subexpresiones más simples, ya que en realidad la computadora va a operar con pares de datos o a veces con datos individuales. La lógica indicaría que la computadora irá ejecutando las subexpresiones de izquierda a derecha, pero no es así, ya que algunas operaciones tiene mayor prioridad que otras, se establecen entonces **reglas de precedencia** que indican que operador va a tener mayor prioridad de resolución sobre otro, y si ambos tuvieran la misma prioridad, se resuelven de izquierda a derecha.

Reglas de precedencia

1. Potenciación (^).
2. Multiplicación (*), división (/), y el resto de la división (%).
3. sumas y restas.
4. operadores relacionales.
5. operador lógico ~ (not).
6. operador lógico & (and).
7. operador lógico | (or).

Estas reglas puede romperse utilizando los **delimitadores** paréntesis "(" y ")" ya que estos delimitadores tienen mayor prioridad sobre cualquier operador.

Veamos un ejemplo tomando la fórmula de cálculo de la distancia recorrida por un proyectil en tiro vertical. En matemática se expresaría como:

$$d = v_i \,.\, t - ½\, g \,.\, t^2$$

donde *d* es la distancia recorrida (la incógnita), v_i es la velocidad inicial del proyectil, *t* es el tiempo que transcurrió y *g* es la aceleración de la gravedad.

Esto en pseudocódigo se expresaría como:

```
d <- vi * t – 1 / 2 * g *  t ^ 2
```

definamos como lote de prueba los siguientes valores:

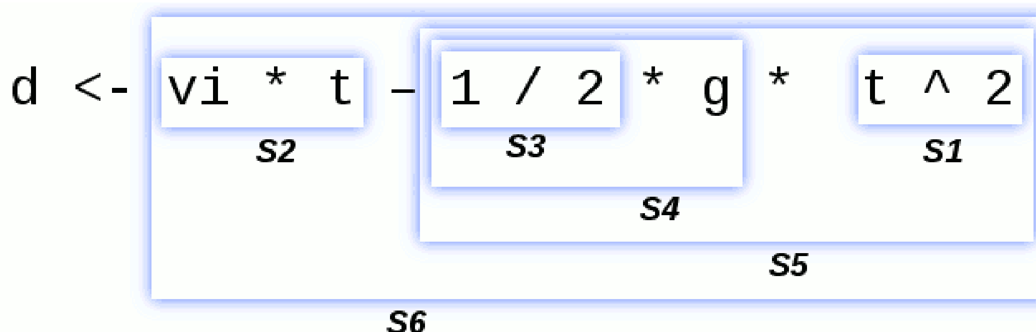
vi ← 100

t ← 10

g ← 9.8

Analicemos por parte, primero d es la variable de destino, no forma parte de la expresión y el símbolo <– es la instrucción que indica la acción de almacenamiento en la memoria.

Luego la expresión está conformada por varias subexpresiones, estas se resuelven según las reglas de precedencia de su operador (se utilizara la letra S para indicar cada subexpresión):



1. Se resuelve primero la potenciación S1 ← t ^ 2 , con lo cual S1 vale 100.
2. Luego hay un conjunto de productos y divisiones en el mismo nivel, por lo tanto tendrá prioridad la de la izquierda: S2 ← v1 * t , (100 * 10) resultado 1000.
3. S3 ← 1 / 2, resultado 0.5
4. S4 ← S3 * g, (0.5 * 9.8) resultado 4.9
5. S5 ← S4 * S1, (4.9 * 100) resultado 490
6. finalmente la resta que posee el menor nivel de precedencia, S6 ← S2 – S5 , (1000 – 490) resultado final 510.

Operadores Lógicos

Vamos a extendernos un poquito más sobre los operadores lógicos **&**, **|** y **~** (AND, OR y NOT).

Estos operadores funcionan según los criterios establecidos en las tablas de verdad del álgebra de Boole. Analicemos el resultado `r` de las expresiones para los 3 operadores, suponiendo que operan sobre dos variables lógicas `v1` y `v2` (en el caso del ~ es solo una por ser monario).

`r ← v1 & v2`

v1	*v2*	*r*
F	F	F
F	V	F
V	F	F
V	V	V

`r ← v1 | v2`

v1	*v2*	*r*
F	F	F
F	V	V
V	F	V
V	V	V

`r ← ~v1`

v1	*r*
F	V
V	F

Delimitadores

Son elementos del lenguaje que cumplen la función de separar o delimitar (de ahí su nombre) datos. Los delimitadores más importantes son:

delimitador	nombre	función
"	comillas	Delimita una constante alfanumérica (literal).
'	apóstrofo	Similar a las comillas, delimita una constante alfanumérica.
( )	paréntesis	Delimita una sub expresión
,	coma	Separa un conjunto de datos
	espacio	Separa palabras
	Salto de línea	Separa instrucciones
[ ]	corchetes	Delimita un sub índice de un arreglo

Comenzando a programar

Vamos a comenzar con algoritmos muy simples y luego los iremos haciendo más complejos para añadir más conceptos.

Comenzaremos con algunos algoritmos lineales y de paso vamos probando la herramienta.

Algoritmo de suma

El enunciado de este problema sería algo como:

> *Se necesita confeccionar un programa que sume dos números y nos muestre el resultado.*

Analicemos los datos que se requieren para resolver el algoritmo.

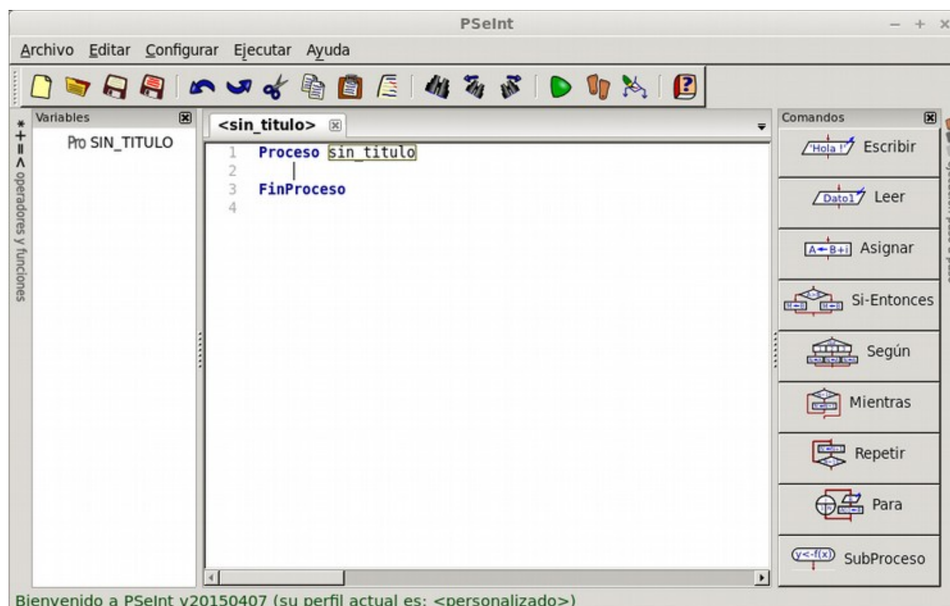
Por un lado necesitamos dos variables para almacenar los dos operandos que proporcionará el usuario, a estos los denominaremos A y B. Necesitaremos además una variable para almacenar el resultado de la suma, a esta la llamaremos C.

Obviamente la expresión a utilizar es A + B.

entonces el algoritmo planteado que da de la siguiente manera:

```
Proceso Suma
    Leer A, B
    C <- A + B
    Escribir "El resultado es: ", C
FinProceso
```

Vamos a comprobar si el algoritmo está bien. Ejecutamos el programa PseInt y nos encontraremos con la siguiente ventana:



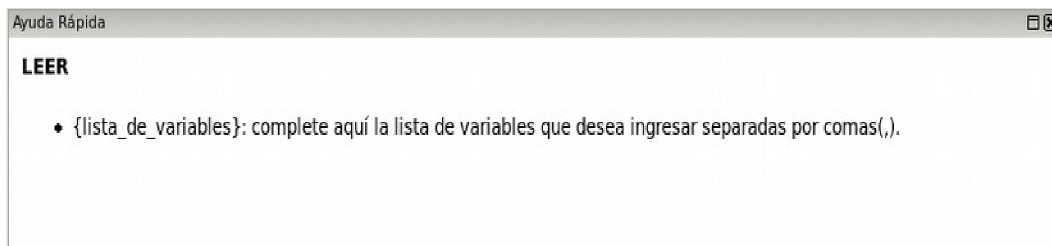
Observemos que nos aparece parte de un pseudocódigo, entre las sentencias `Proceso` y `FinProceso` deberemos escribir nuestro código, pero mejor será

utilizar los botones que se ven al costado derecho en la barra cuyo nombre es `Comandos`.

Lo primero que debemos poner es un ingreso de datos, por lo tanto debemos utilizar el botón `Leer` que tiene el diagrama: Dato1 . al presionarlo obtenemos la siguiente línea de texto:

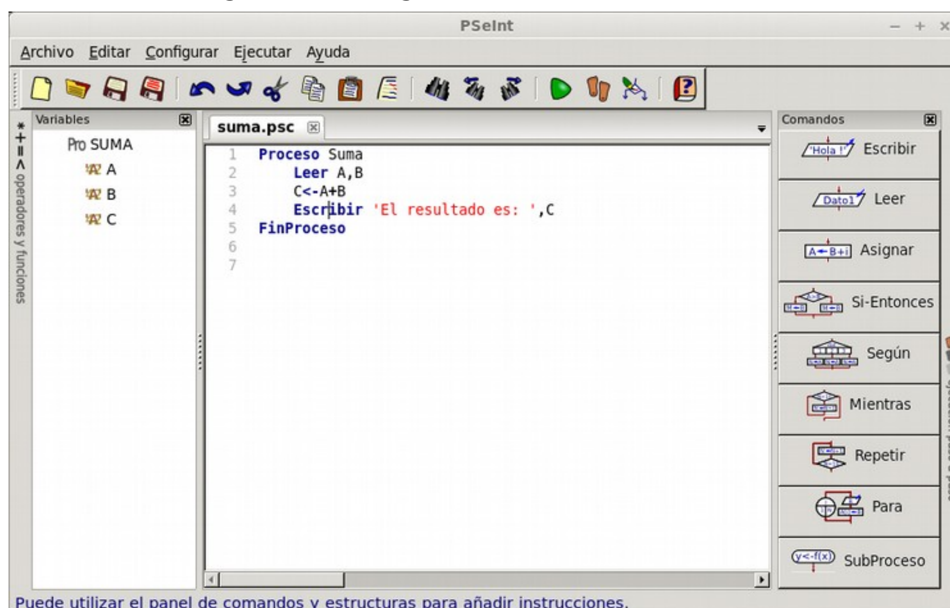
```
Leer lista_de_variables
```

Ahora debemos reemplazar el recuadro que dice "Lista de Variables" por nuestras variables separadas por comas, es lo que nos sugiere en el cuadro inferior que apareció al presionar el botón.



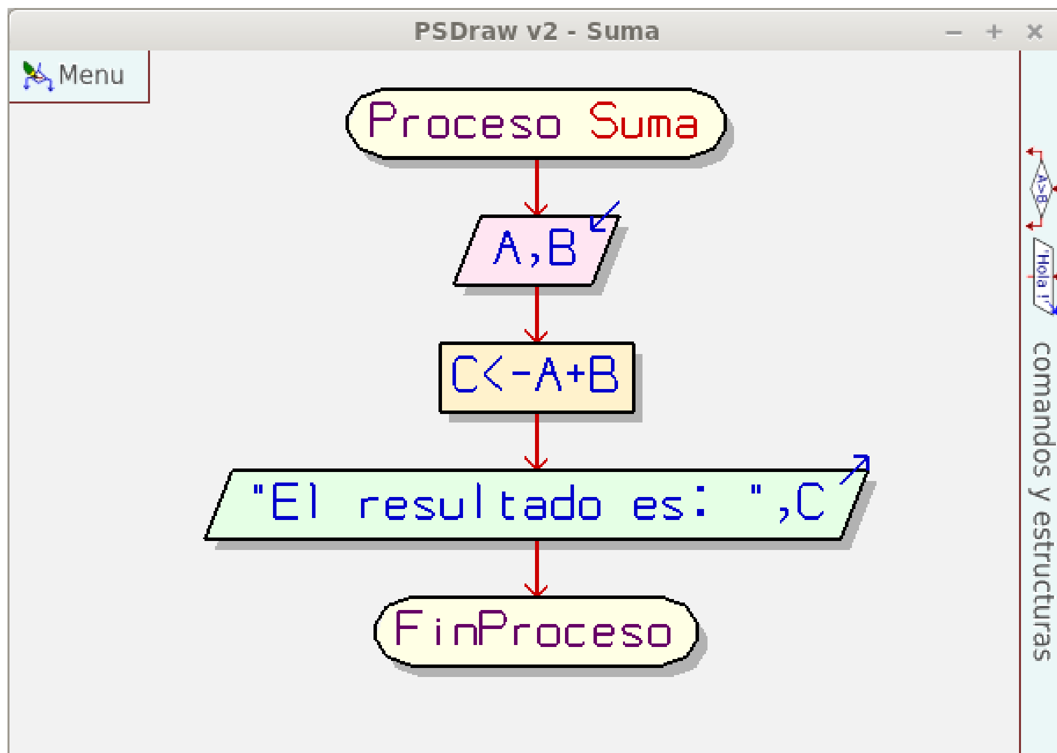
Como vemos, está toda la línea grisada, para seleccionar solamente el cuadro que dice "Lista de Variables" nos movemos con las flechas de cursor hacia atrás y luego hacia adelante hasta que se seleccione solamente lo que queremos. Una vez seleccionado el cuadro de lista debemos tipear nuestras variables según el algoritmo diseñado. Repetimos las acciones con los botones de asignar y Escribir con lo que debemos obtener el pseudocódigo completo. Podemos también cambiar el nombre del proceso, le pondremos de nombre "Suma".

Antes de continuar es conveniente grabar el pseudocódigo generado, presionamos el ícono y nos aparecerá un cuadro de diálogo preguntando el nombre del archivo y donde guardarlo. Una vez hecho esto, la ventana quedará como la vemos en la siguiente imagen:



Ahora podemos cambiar a vista de Diagrama de Flujo, simplemente presionando el botón con el siguiente ícono en la barra de botones nos

abrirá una ventana donde podremos apreciar como queda el diagrama de flujo de datos, también lo podremos exportar como imagen:



Ahora que el algoritmo está terminado, probaremos mediante PseInt el funcionamiento del algoritmo al ser ejecutado por la computadora. Presionamos ahora el ícono de la barra de botones y se nos abrirá una nueva ventana:

La

PSeInt - Ejecutando proceso SUMA

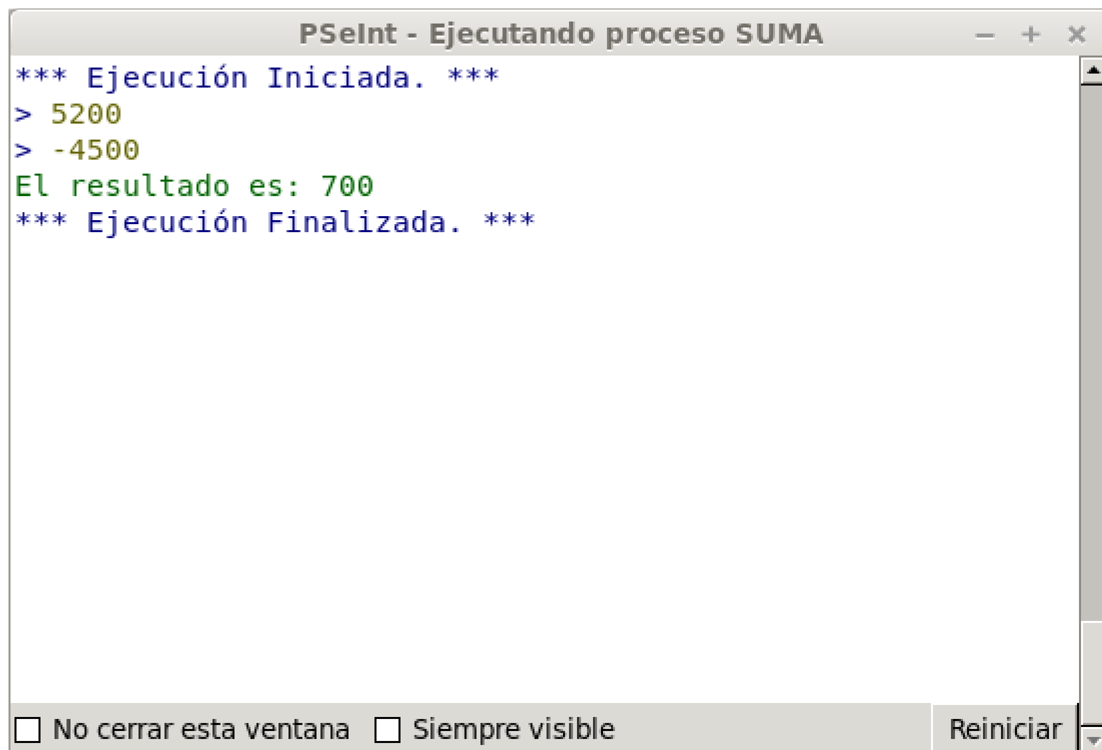
*** Ejecución Iniciada. ***

>

linea 2 instruccion 1

idea es que ingresemos ambas variable que deberán ser sumadas, por ejemplo

tomaremos como lote de prueba los valores 5200 y -4500, si el algoritmo está correcto el resultado debería ser 700... veamos: Ingresamos primer o el valor 5200 y presionamos intro (o enter), luego el valor -4500 y de nuevo Intro. Inmediatamente nos muestra el resultado que efectivamente era lo que esperábamos:



Digamos que este algoritmo fue diseñado solo con lo esencial para ver si funciona, pero como lo estamos ejecutando vemos que podría ser un poco más amigable, entonces reformulemos un poquito las instrucciones del pseudocódigo:

```
Proceso Suma
    Escribir "A continuación ingrese dos valores para sumarlos"
    Leer A,B
    C<-A+B
    Escribir 'El resultado es: ',C
FinProceso
```

Eso ayudará a intuir para qué sirve el algoritmo si lo compartimos con algún usuario.

Algoritmo del triangulo.

Veamos como encarar un problema un poco más complejo. El enunciado sería:

> *Calcular el área y el perímetro de un triángulo conociendo el largo de los 3 lados.*

Parece simple no? Bueno, si es simple, por lo menos para calcular perímetro = lado + lado + lado... Sí... pero hay un problema, seguramente recordamos que el área de un triangulo se calcula con la fórmula: área = (base x altura) / 2.

En el caso que los datos ingresados correspondan a un triángulo rectángulo, no habría problemas simplemente habría que identificar que lado corresponde a la base y cual a la altura, ¿pero si no es un triángulo rectángulo? Habría que calcular el valor altura... ¿y cómo se calcula? Bueno, podemos ir a un buscador de la web y consultar un sitio que nos diga "cómo se calcula el área de un triángulo escaleno". El buscador nos llevaría a varios sitios donde encontraremos el "teorema de Herón de Alejandría" que dice: «*En un triángulo de lados a, b, c, y semiperímetro s=(a+b+c)/2, su área es igual a la raíz cuadrada de s(s-a)(s-b)(s-c).*» (sacado de la wikipedia). Bueno, el teorema habla de cualquier tipo de triángulo, justo es lo que necesitábamos. A esto nos referimos por **comprensión del problema**.

Entonces ya podemos definir las variables:

- Variables de entrada: los 3 lados que llamaremos L1, L2 y L3
- Variables de salida: el área que la llamaremos A y el perímetro que lo llamaremos P.
- Variables de proceso: Según el teorema de Herón, necesitamos calcular el semiperímetro, al que llamaremos S.

las fórmulas a aplicar serían:

P = L1 + L2 + L3

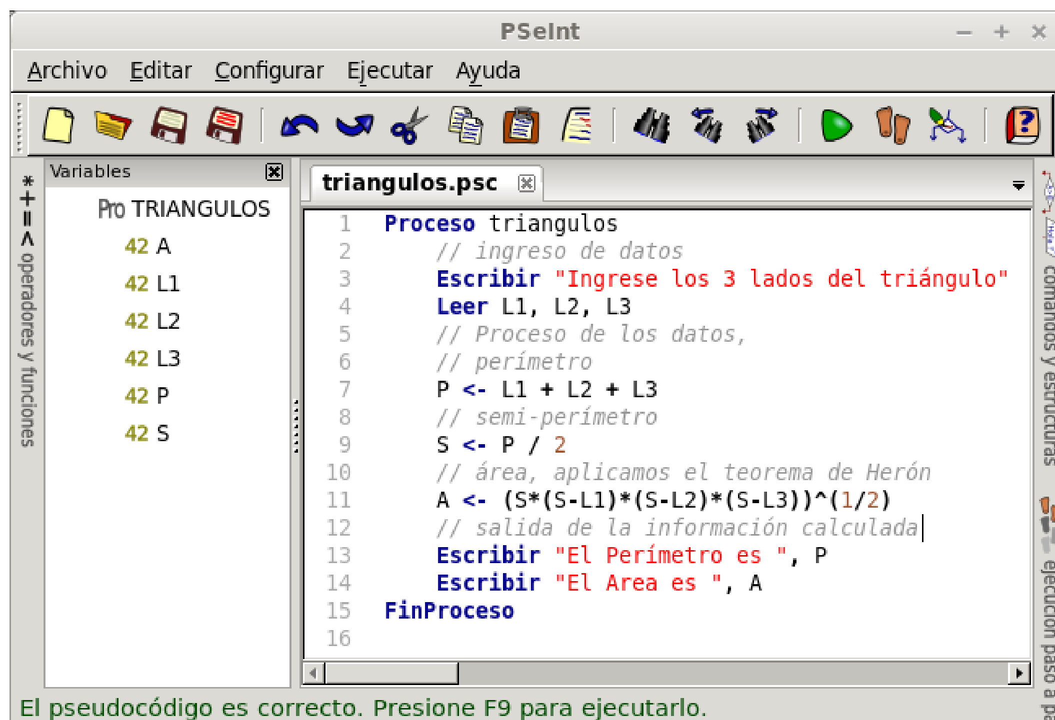
S = P / 2

A = $\sqrt{S \times (S - L1) \times (S - L2) \times (S - L3)}$

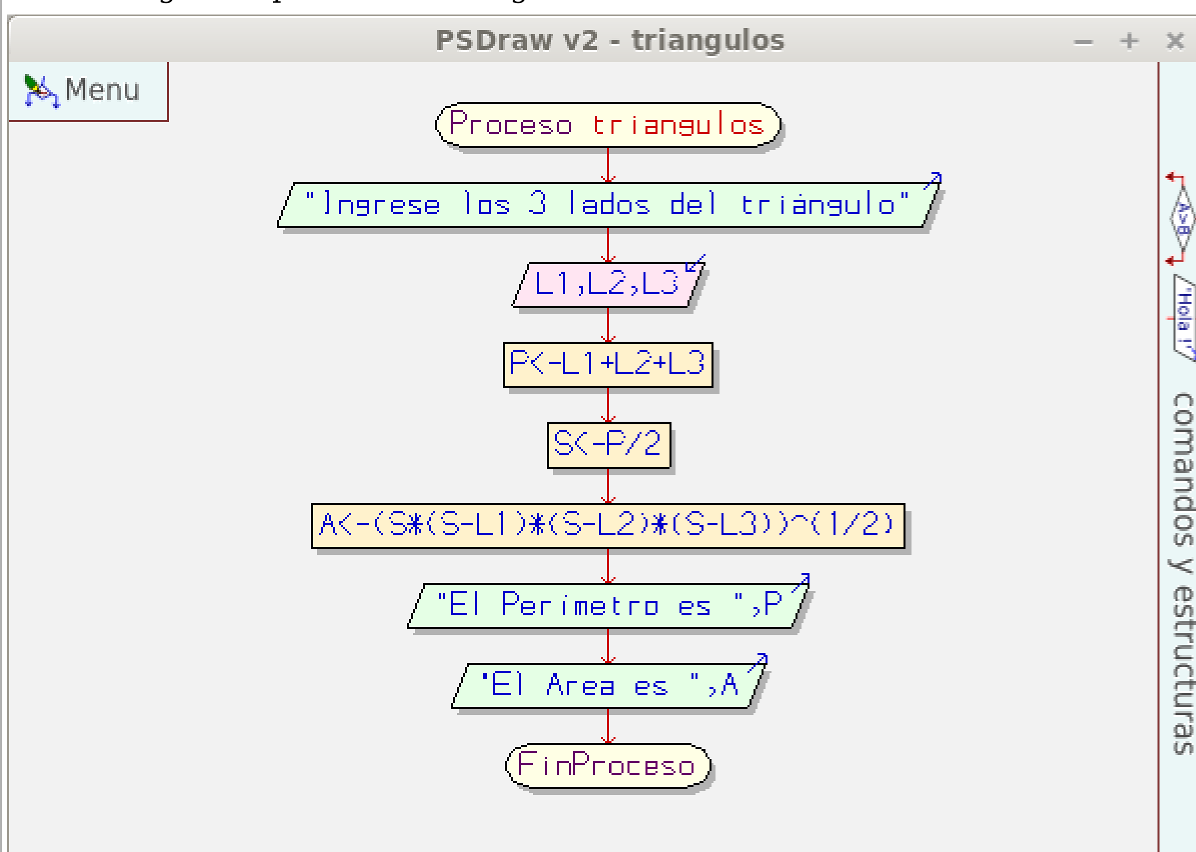
Diseñemos entonces el algoritmo utilizando pseudocódigo y el DFD. Pero primero haremos un par de observaciones:

Para obtener la raíz cuadrada recurrimos a las matemáticas, la raíz cuadrada de un número es equivalente a elevar ese número a ½.

Por otra parte, agregaremos comentarios al pseudocódigo para recordar en un futuro de donde sacamos la fórmula. El programa PseInt admite el añadido de comentarios al pseudocódigo utilizando una doble barra (//) al principio del texto. Esta forma de comentar es la que se utiliza en el lenguaje de programación "C++" y en muchos de los lenguajes basados en este, como Java, PHP, Javascript, etc.



Y el diagrama quedaría de la siguiente forma:



Comenzando con la programación estructurada

Sigamos complicando el algoritmo del triángulo.

Pensemos lo siguiente: sabemos que si hablamos de la longitud de los lados hacemos referencia a un valor positivo mayor que 0, eso es obvio... pero no necesariamente lo será para un usuario. ¿Qué pasaría si un usuario por falta de comprensión ingresara un valor negativo o cero? Pues, entonces el algoritmo informaría como válido un resultado erróneo.

Estructura de Decisión

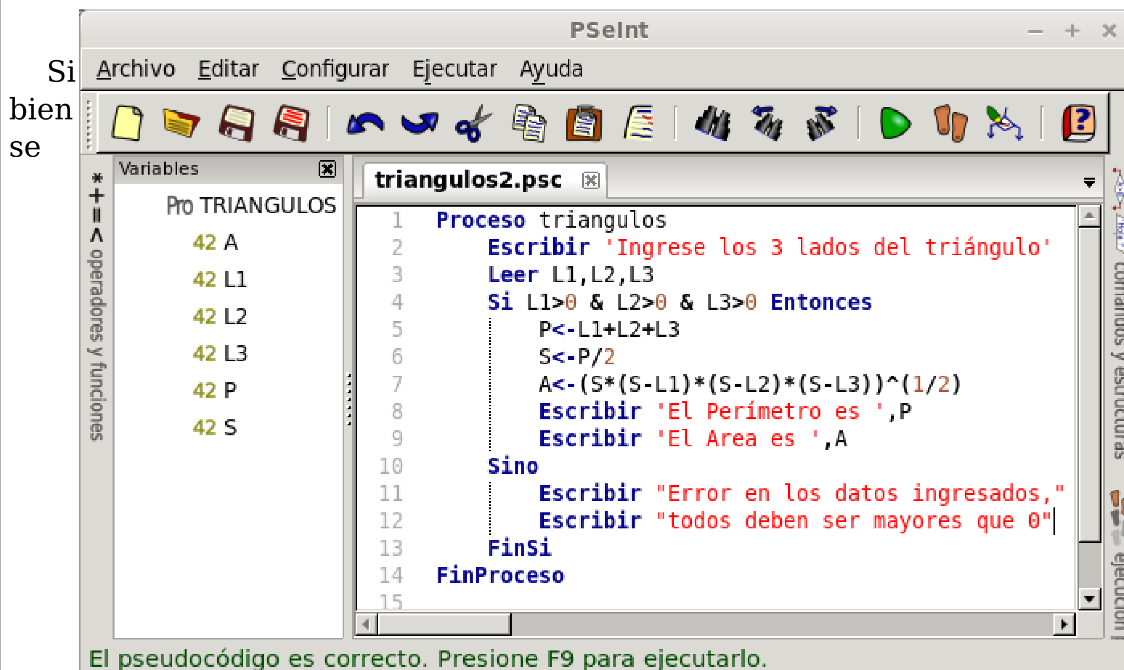
Podemos entonces validar si el usuario ingresó valores permitidos. Simplemente evaluando por cada ingreso que este sea mayor a 0. Para validar utilizamos una **estructura de decisión** que nos permitirá bifurcar el algoritmo en dos ramas, una que será la válida y otra que será inválida en la cual informaremos al usuario de su error.

La estructura de decisión se escribe en pseudocódigo como:

```
Si <expresión lógica> Entonces
 <bloque de acciones por verdadero>
Sino
 <bloque de acciones por falso>
Fin Si
```

Si bien ya lo mencionamos, reiteraremos que la expresión lógica mencionada tendrá como resultado verdadero o falso. Si es verdadero se ejecutará el primer bloque de instrucciones y si es falso se ejecutará el segundo, nunca se ejecutarán ambos.

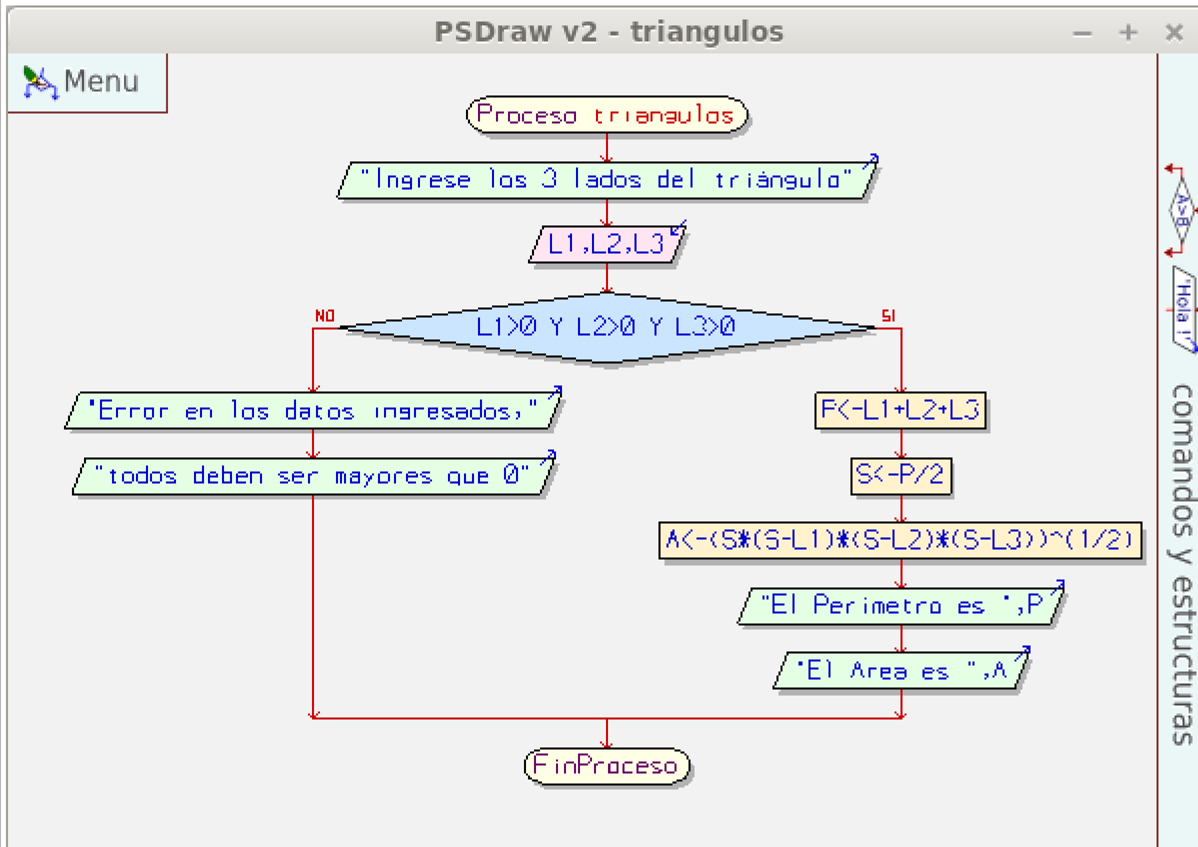
Como quedará el pseudocódigo entonces?



Si bien se

podría aplicar una estructura de decisión para evaluar cada valor, utilizamos una expresión que combina la evaluación de todos los valores a la vez, la expresión se debe entender como: si todas las subexpresiones (L*n* > 0) son verdaderas entonces por efecto del operador AND la expresión completa será verdadera, si alguna subexpresión es falsa, la expresión completa será entonces falsa.

Veamos el diagrama:



Estructura Repetitiva

Sigamos pensando sobre el mismo algoritmo.

En el caso que el usuario se equivoque, entonces, ¿deberá ejecutar nuevamente el algoritmo desde cero? Por ahora la respuesta es: Sí, deberá comenzar de nuevo con el algoritmo. ¿Y qué pasa si quiero conocer el perímetro y la superficie de varios triángulos? ¿No habrá una forma de que me siga permitiendo calcular las veces que yo quiero sin tener que ejecutar nuevamente el algoritmo?. O mejor... No se podrá modificar el algoritmo para que me indique cual es el triangulo de mayor área de un grupo de triángulos que quiero procesar? Bueno.. ya es pedir mucho... reformulemos primero el enunciado del problema.

> *Calcular el área y el perímetro de varios triángulos conociendo el largo de los 3 lados y determinar cual de ellos tiene el área mayor.*

En primer lugar evaluemos que significa calcular “varios” triángulos, eso implica que debemos repetir el cálculo una y otra vez a medida que se ingresan

los 3 lados. Para esto utilizaremos la estructura repetitiva que en pseudocódigo se escribe cómo:

```
Mientras <expresión_logica> Hacer
 <bloque de instrucciones>
Fin Mientras
```

La expresión lógica es la que determina si se volverá a ejecutar el bloque de instrucciones mientras sea de resultado verdadero, cuando la expresión tiene como resultado falso entonces ya no se ejecutará más el bloque y se seguirá con la secuencia normal.

Pero para encarar el nuevo algoritmo debemos agregar a las incógnitas conocidas y analizadas, una nueva variable que será el mayor valor de área calculado. Esta variable no se resuelve en forma directa como el área y el perímetro. Si tuviéramos que resolver el problema del mayor área con lápiz y papel seguramente haríamos todos los cálculos de todas las áreas y luego al finalizar revisaríamos todo para encontrar el mayor, el problema es que el algoritmo diseñado no recuerda los valores ingresados, tan sólo el último procesado. Pero hay un método mejor para aplicarlo al modo de cálculo de la computadora según conocemos hasta el momento. Esta forma requeriría que cada vez que hagamos un cálculo comparemos si el área es la mayor encontrada hasta ese momento, en este caso cuando lleguemos al final estaremos seguros que hemos comparado cada área a medida que la calculábamos. Además, será mejor que guardemos cuales eran los lados ingresados para recordarle al usuario al finalizar el proceso.

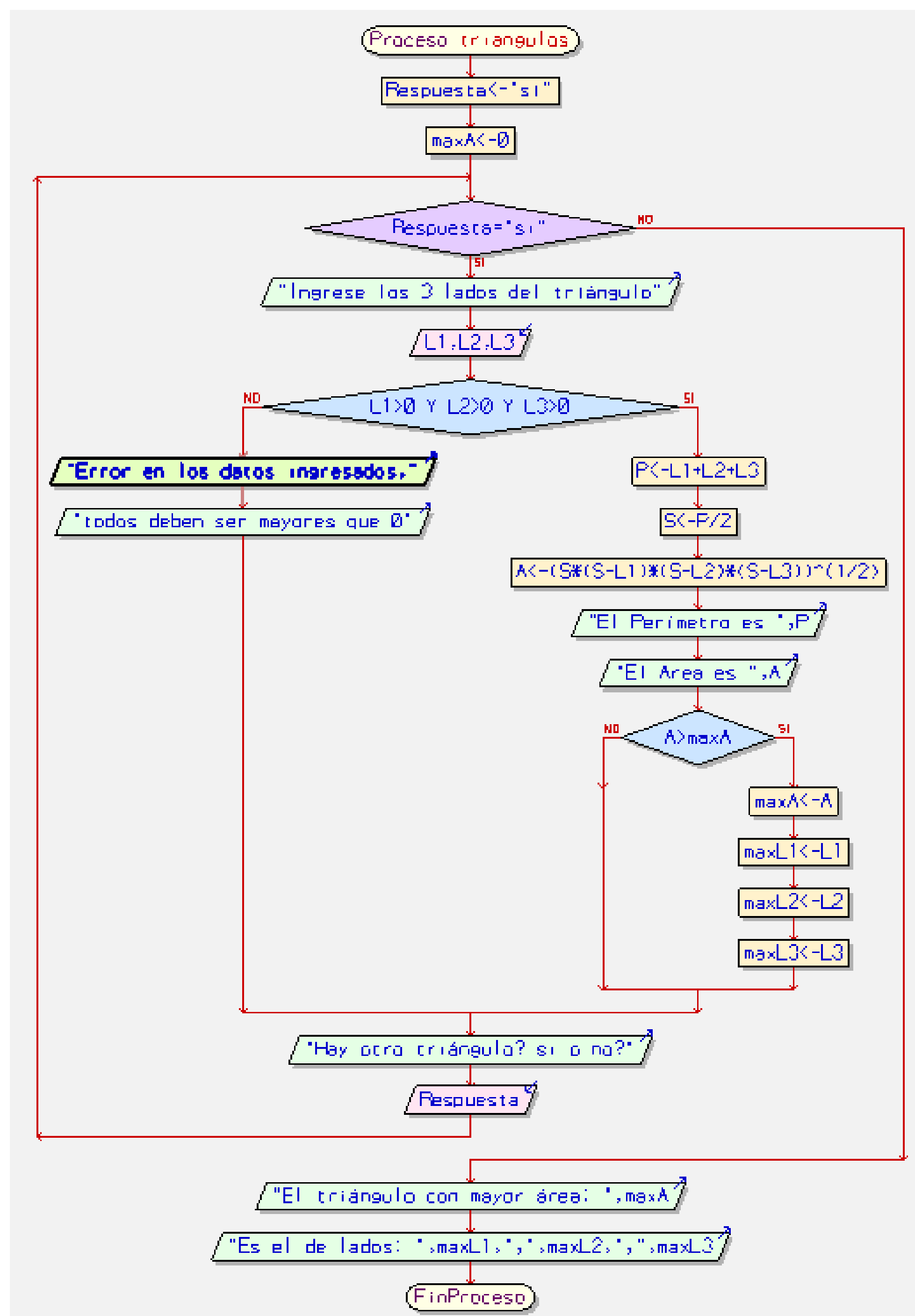
El otro tema que falta resolver es darnos cuenta cuando no hay más triángulos que procesar, una forma sería preguntar al usuario si hay más triángulos, cuando él diga que no entonces se dejará de calcular y se podrá mostrar el mayor.

Entonces el algoritmo en pseudocódigo quedaría de la siguiente forma:

```
Proceso triangulos
    Respuesta <- "si" //obligo a ingresar la primera vez
    maxA <- 0         // Area maxima (0 es la cota inferior)
    Mientras Respuesta = "si" Hacer
        Escribir 'Ingrese los 3 lados del triángulo'
        Leer L1,L2,L3
        Si L1>0 & L2>0 & L3>0 Entonces
            P<-L1+L2+L3
            S<-P/2
            A<-(S*(S-L1)*(S-L2)*(S-L3))^(1/2)
            Escribir 'El Perímetro es ',P
            Escribir 'El Area es ',A
            Si A > maxA Entonces // determino si el área es máxima
                maxA <- A        // guardo el nuevo máximo
                maxL1 <- L1      // y guardo los lados L1, L2 y L3
                maxL2 <- L2
                maxL3 <- L3
            Fin Si
        Sino
```

```
				Escribir "Error en los datos ingresados,"
				Escribir "todos deben ser mayores que 0"
			FinSi
			Escribir "Hay otro triángulo? si o no?"
			Leer Respuesta // si o no
		Fin Mientras
		Escribir "El triángulo con mayor área: ", maxA
		Escribir "Es el de lados: ", maxL1, ",", maxL2, ",", maxL3
	FinProceso
```

y su respectivo diagrama:



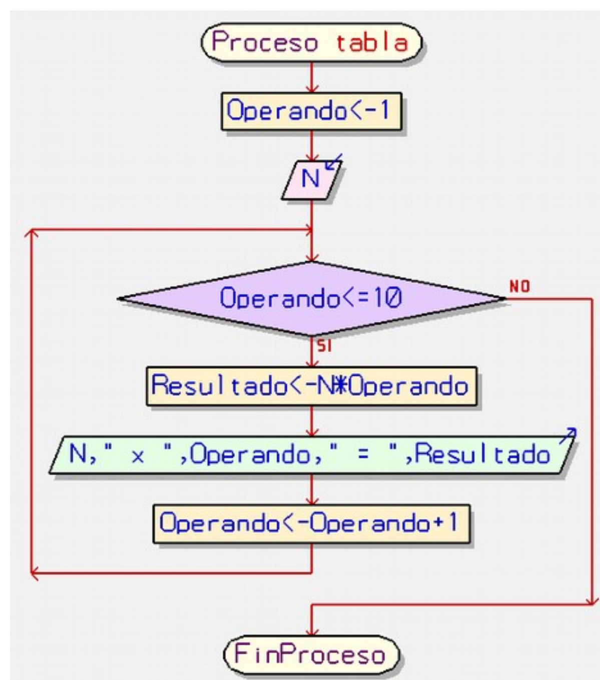
Algoritmo de la Tabla de Multiplicar

Hagamos ahora otro algoritmo. Planteemos el enunciado primero:

Confeccionar un algoritmo que nos muestre por pantalla la tabla de multiplicar de un número ingresado.

El enunciado es mas bien simple de entender, tenemos una variable que será ingresada, digamos `N`, luego para hacer la tabla del número N tendremos que multiplicarla por todos los números desde 1 hasta 10, para esto utilizaremos una variable que funcionará como **contador** al que llamaremos variable `Operando`. El algoritmo también es bastante simple a esta altura.

```
Proceso tabla
    Operando <- 1
    Escribir "Ingrese el número para la tabla de multiplicar"
    Leer N
    Escribir "Tabla del Número ", N
    Mientras Operando <= 10 Hacer
        resultado <- N * Operando
        Escribir N, " x ", Operando, " = ", Resultado
        Operando = Operando + 1
    FinMientras
    Escribir "--------------------"
FinProceso
```



Estructura Iterativa

Para los casos en que se debe controlar el ciclo mediante una variable de tipo contadora existe una estructura particular que puede sustituir con ventaja a la estructura anterior. Nos referimos a la **estructura iterativa**, la cual se asocia a una variable que se incrementa automáticamente y que al llegar a un

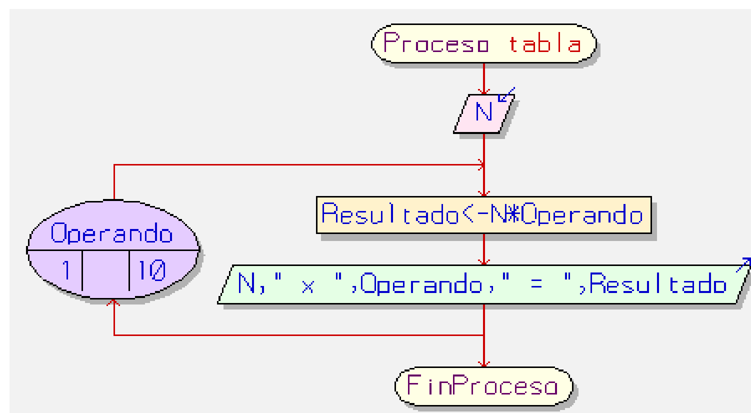
determinado valor interrumpe las repeticiones del bloque.

Utilizando pues este tipo de estructura el algoritmo quedaría planteado de la forma siguiente:

El

```
Proceso tabla
    Escribir "Ingrese el número para la tabla de multiplicar"
    Leer N
    Escribir "Tabla del Número ", N
    Para Operando<-1 Hasta 10 Hacer
        resultado <- N * Operando
        Escribir N, " x ", Operando, " = ", Resultado
    Fin Para
    Escribir "--------------------"
FinProceso
```

Diagrama de Flujo de Datos quedaría de la siguiente forma (para simplificar quitamos los mensajes referenciales).



¿Qué beneficios concretos obtenemos de utilizar esta estructura? Bueno, todos pueden ser discutibles ya que no todos los beneficios que mencionaremos son objetivos, pero diremos:

- No hace falta controlar la variable contadora, ya que el control será automático.
- No hace falta incrementar la variable contadora, también se hará automáticamente, ahorrando una línea de código.
- Sobre todo en el caso que se utilicen muchos ciclos iterativos, es mucho más claro el enunciado del algoritmo.
- Cuando se traduzca el algoritmo a un lenguaje de programación este será capaz de optimizar los ciclos utilizando variables de tipo registro (un registro dentro de la Unidad de Control es mucho más rápido de utilizar que una variable alojada en la RAM).

Variables especiales

Anteriormente mencionábamos una variable de función **contadora**, esta es una variable numérica normal que tiene una función especial, pero además de esta hay otras, las analizaremos.

Variable contadora

Una variable contadora tiene la función de contar ocurrencias en un algoritmo, ¿qué es esto de contar ocurrencias? Analicemos un ejemplo comenzando con un enunciado:

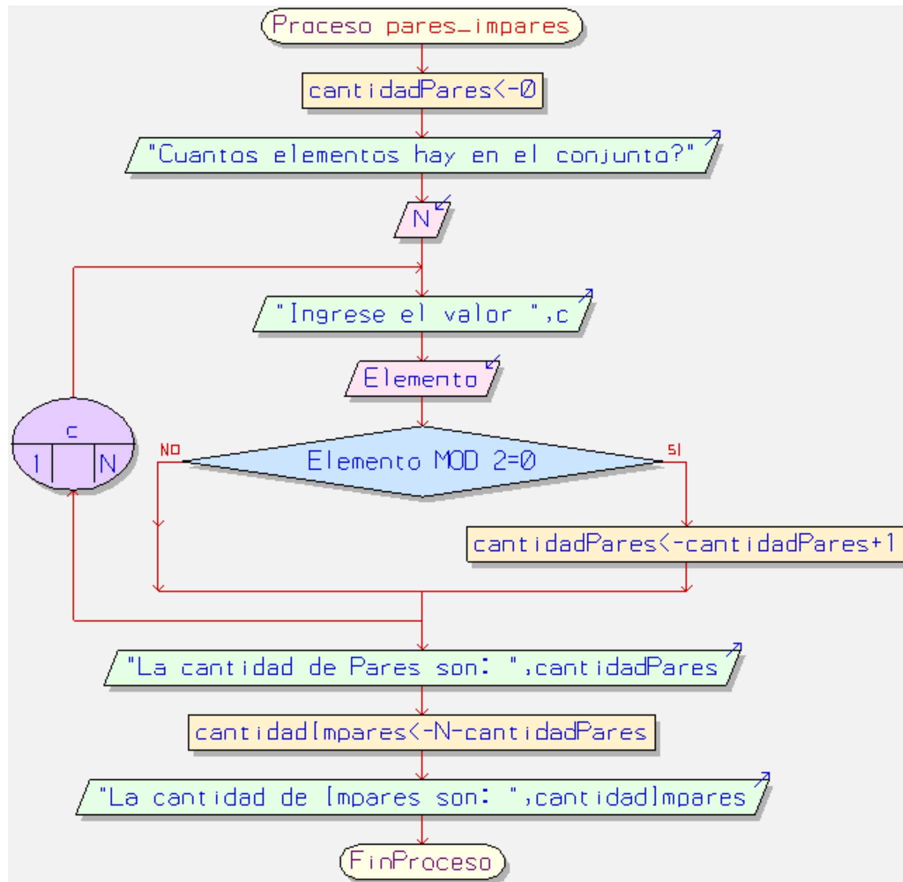
> *De un conjunto de N elementos numéricos enteros, indicar cuantos son pares y cuantos son impares.*

Ideamos el algoritmo: debemos hacer que el usuario primero nos defina cuantos valores existen en el conjunto (el valor de N), luego los debería ingresar uno por uno en un ciclo de N veces, y a medida que los vaya ingresando hay que verificar si el valor ingresado es par y en el caso que lo sea, contarlo. Al final se debe mostrar el contador que indica los pares contados y si a N le restamos el contador, obtenemos la cantidad de impares.

Hagamos el pseudocódigo

```
Proceso pares_impares
    cantidadPares <- 0
    Escribir "Cuantos elementos hay en el conjunto?"
    Leer N
    Para c <- 1 Hasta N Hacer
        Escribir "Ingrese el valor ", c
        Leer Elemento
        Si Elemento % 2 = 0 Entonces
            cantidadPares <- cantidadPares + 1
        FinSi
    FinPara
    Escribir "La cantidad de Pares son: ", cantidadPares
    cantidadImpares <- N - cantidadPares
    Escribir "La cantidad de Impares son: ", cantidadImpares
FinProceso
```

Y el diagrama de flujo:



Observemos la variable contadora denominada `cantidadPares` que es la que cuenta los elementos pares. Tiene tres estados,

- El primero es la inicialización: al principio del algoritmo recibe un valor inicial que generalmente es 0. Es el valor de partida del contador.
- El segundo estado es dentro del ciclo: cuando es incrementado aplicando la instrucción típica `contador ← contador + k`, donde k es un valor constante (se puede contar de a dos o de a docena, por ejemplo), aunque lo más normal es que k sea equivalente a 1.
- El tercer estado ocurre al salir del ciclo, cuando disponemos de la variable con su valor definitivo y por tanto podemos como en el caso presente, mostrarla por pantalla o utilizarla para los cálculos de totales.

Variable acumuladora

Pero que pasa si en lugar de sumarle valores constantes queremos sumarle valores variables. Analicemos un nuevo enunciado de un problema.

Queremos obtener el promedio de edad de un grupo de personas.

Para calcular el promedio la expresión a aplicar sería: la suma total de las edades sobre la cantidad de personas. Dicho de otra manera sería:

`promedio ← acumulador / contador`.

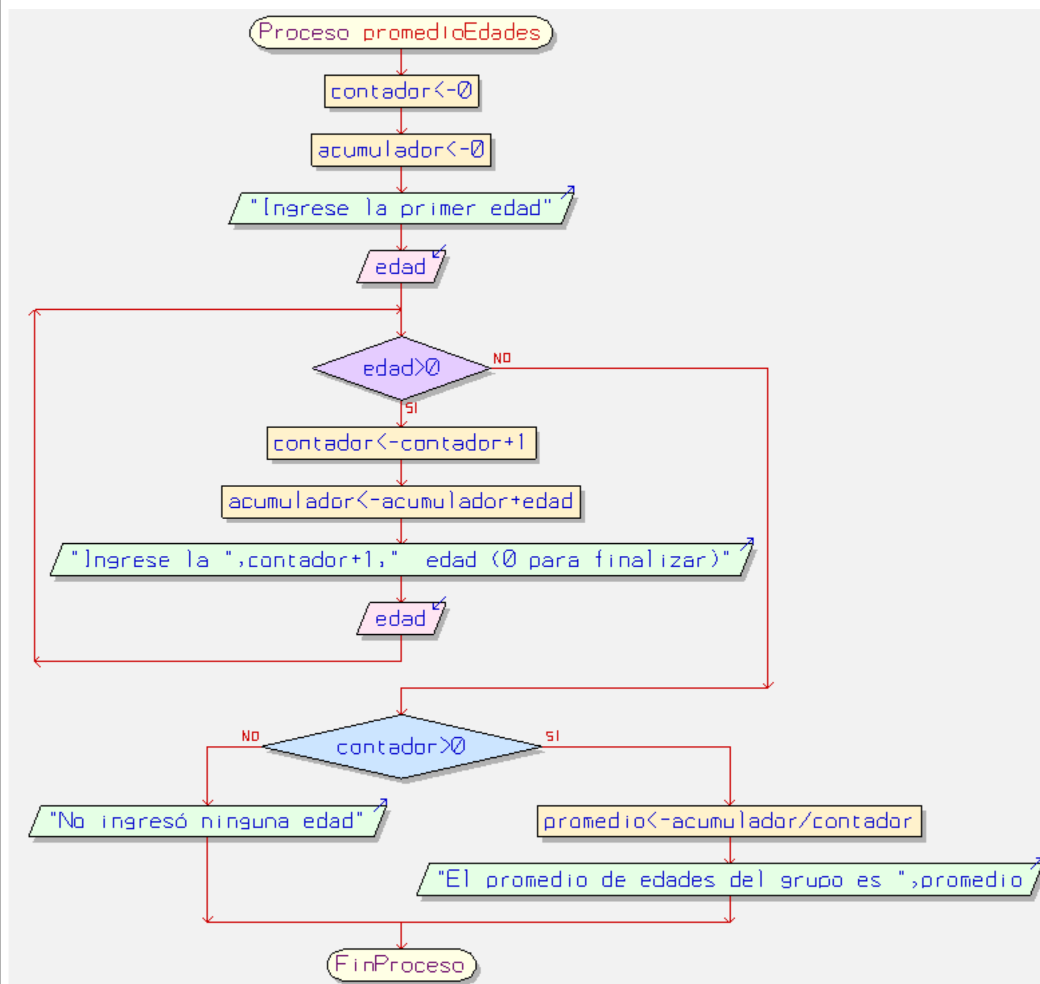
Ahora vamos a considerar como debe hacer el usuario para indicarnos que la carga de datos ha finalizado, una forma es preguntarle cada vez que se termina

de procesar una edad si existen más personas sin procesar, es bastante incómodo tener que escribir “si” cada vez... Otra forma, que vamos a utilizar en este caso, es indicarle que ingrese una edad imposible cuando ya no hay más personas. En este caso podemos decirle que al ingresar 0 se entiende que finalizó la carga de datos.

El algoritmo quedaría de la siguiente manera:

```
Proceso promedioEdades
    contador <- 0
    acumulador <- 0
    Escribir "Ingrese la primer edad"
    Leer edad
    Mientras edad > 0 Hacer
        contador <- contador + 1
        acumulador <- acumulador + edad
        Escribir "Ingrese la ", contador+1, "º edad (0 para finalizar)"
        Leer edad
    Fin Mientras

    Si contador > 0 Entonces
        promedio <- acumulador / contador
        Escribir "El promedio de edades del grupo es ", promedio
    Sino
        Escribir "No ingresó ninguna edad"
    Fin Si
FinProceso
```



Variable flag, bandera o señal

Esta variable es generalmente una variable de tipo lógico, que indica cuando ha sucedido un evento mediante un cambio de estado. Vamos a ver un ejemplo para entenderlo mejor, planteemos el enunciado de un problema:

> *En un conjunto de elementos numéricos enteros sin el cero (pueden existir positivos y negativos) encontrar el menor de los elementos que sea divisible por el número 3.*

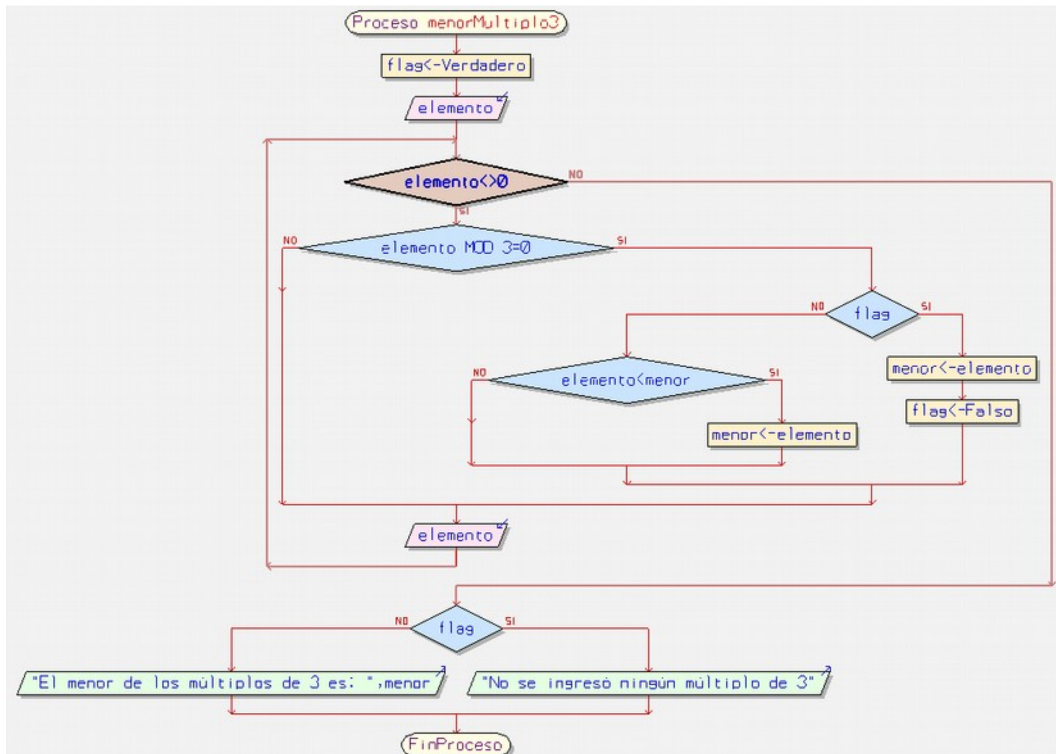
Aquí existen elementos de cualquier valor, excepto el cero que fue dejado fuera del conjunto adrede, para simplificar el algoritmo y utilizarlo como señal. Aquí vemos el uso de la variable señal, en este caso está digitada en forma externa.

Pero vamos a tener que agregar una señal más para saber cuando inicializar la variable que va a contener el menor. ¿por qué? Bueno, sabemos que pueden o no existir valores negativos pero no estamos seguro de eso, por lo tanto no podemos utilizar el cero como valor inicial para la variable menor ya que al comparar, si todos lo valores son positivos nunca se encontrará uno menor que cero. También esta señal nos servirá para determinar si existe al menos un número divisible por 3, porque también puede pasar que no exista ningún múltiplo. Entonces con este flag inicializaremos la variable menor con el primer múltiplo de 3 que se ingrese.

El

```
Proceso menorMultiplo3
    flag <- Verdadero
    Leer elemento
    Mientras elemento <> 0 Hacer
        Si elemento % 3 = 0 Entonces
            Si flag Entonces
                menor = elemento
                flag = Falso
            Sino
                Si elemento < menor Entonces
                    menor = elemento
                Fin Si
            Fin Si
        Fin Si
        Leer elemento
    Fin Mientras
    Si flag Entonces
        Escribir "No se ingresó ningún múltiplo de 3"
    Sino
        Escribir "El menor de los múltiplos de 3 es: ", menor
    Fin Si
FinProceso
```

Diagrama de Flujo de Datos:



Intercambio de variables o swap

El intercambio de variables es una operación que se realiza mucho cuando se trabaja con grandes volúmenes de datos, por ejemplo para realizar un ordenamiento de valores. Consiste en hacer que el contenido de una variable se traslade a otra variable al mismo tiempo que el contenido de la segunda variable pasa a la primera. El proceso es muy simple pero requiere de una variable auxiliar que almacene temporalmente uno de los valores para no perderlo. Veamos el pseudocódigo:

```
Proceso swap
    A <- 18
    B <- 35
    Mostrar "Valor de A: ", A
    Mostrar "Valor de B: ", B
    auxiliar <- A
    A <- B
    B <- auxiliar
    Mostrar "Luego del intercambio:-----------------------"
    Mostrar "Valor de A: ", A
    Mostrar "Valor de B: ", B
FinProceso
```

```
PSeInt - Ejecutando proceso SWAP
*** Ejecución Iniciada. ***
Valor de A: 18
Valor de B: 35
Luego del intercambio:-----------------------
Valor de A: 35
Valor de B: 18
*** Ejecución Finalizada. ***
No cerrar esta ventana   Siempre visible   Reiniciar
```

Funciones

Vamos a analizar todas las funciones soportadas por PseInt en su versión 20150407, pero primero definamos en qué consiste una función.

Una función es un procedimiento definido por el lenguaje de programación que permite obtener un resultado a través del proceso de uno, varios o ningún dato denominados parámetros. Por ejemplo en la instrucción y ← f(x) donde f es una función, x es el parámetro y el resultado es almacenado en la variable y.

Matemáticas

RAIZ(Número) o RC(Número)

Obtiene la Raíz Cuadrada del valor de Número.

ABS(Número)

Obtiene el Valor Absoluto (valor con signo positivo) del valor de Número.

TRUNC(Número)

Trunca el valor de Número obteniendo sólo la parte entera.

REDON(Número)

Devuelve el entero más cercano al valor de Número.

AZAR(Número)

Genera un valor aleatorio entero entre 0 y Número-1 incluidos.

LN(Número)

Obtiene el Logaritmo Natural del valor de Número

EXP(Número)

Obtiene la Función Exponencial (y <- ex) del valor de Número

Trigonométricas

SEN(Número)

Obtiene el Seno del valor de Número tomado en radianes.

COS(Número)

Obtiene el Coseno del valor de Número tomado en radianes.

TAN(Número)

Obtiene la Tangente del valor de Número tomado en radianes.

ASEN(Número)

Obtiene el Arcoseno en radianes del valor de Número.

ACOS(Número)

Obtiene el Arcocoseno en radianes del valor de Número.

ATAN(Número)

Obtiene el Arcotangente en radianes del valor de Número.

Funciones de cadena de texto

LONGITUD(Cadena)

Devuelve la cantidad de caracteres que componen la Cadena.

MAYUSCULAS(Cadena)

Retorna una copia de la Cadena con todos sus caracteres en mayúsculas.

MINUSCULAS(Cadena)

Retorna una copia de la Cadena con todos sus caracteres en minúsculas.

SUBCADENA(Cadena, Número1, Número2)

Retorna una nueva cadena que consiste en la parte de la Cadena que va desde la posición Número1 hasta la posición Número2 (incluyendo ambos extremos). Las posiciones utilizan la misma base que los arreglos, por lo que la primer letra será la 0 o la 1 de acuerdo al perfil del lenguaje utilizado.

CONCATENAR(Cadena1, Cadena2)

Retorna una nueva cadena resulta de unir las Cadenas 1 y 2.

CONVERTIRANUMERO(Cadena)

Recibe una cadena de caracteres que contiene un grupo de caracteres numéricos y devuelve una variable numérica con el valor equivalente.

CONVERTIRATEXTO(Número)

Recibe un real y devuelve una variable numérica con la representación como cadena de caracteres de dicho real.

Arreglos

Un Arreglo o array es un conjunto de variables que tiene las siguientes características:

- Todas las variables comparten un mismo nombre.
- Cada variable se identifica además de por el nombre, por una posición numérica (llamada índice) por cada dimensión definida.
- Todas las variables están ordenadas en forma secuencial a partir de 0 o 1 dependiendo de la configuración del programa PseInt. Para los ejemplos estableceremos que la numeración comienza en 1.
- Son homogéneas, todas las variables son del mismo tipo.

Lo importante de este tipo de variables son las facilidades que brinda para procesar un conjunto de datos.

Dimensión e índice

A qué nos referimos con dimensión? En el caso de las variables las dimensiones son convenciones lógicas, pero podemos hacer un paralelismo con las dimensiones físicas para comprender el concepto.

Simbolicemos una variable como un espacio de almacenamiento en memoria pero démosle una estructura física como si fuese un cajón portaobjetos.

Hasta ahora hemos tratado a las variables como elementos individuales, como si fuera un punto en un espacio o sea de dimensión 0 o sin dimensión. Entonces para reconocer a la variable simplemente se la referenciaba por su nombre. Por ejemplo a esta variable la llamaremos caja.

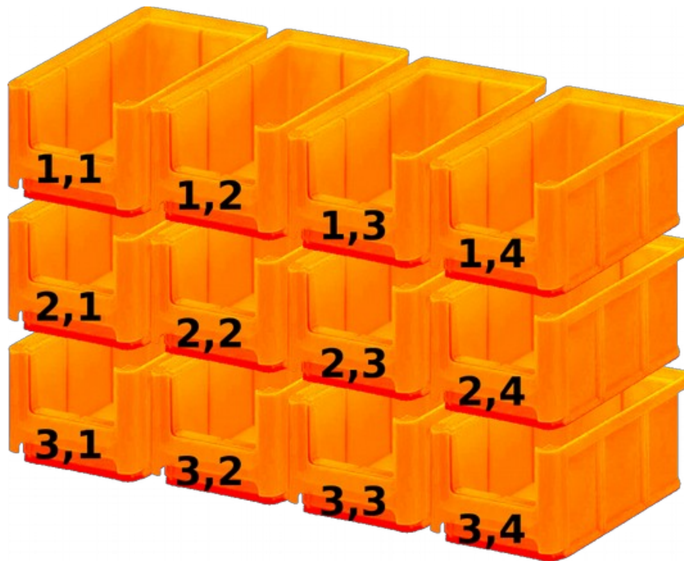


Luego podemos considerar a un arreglo de dimensión uno equivalente a una estructura física de una sola dimensión, o sea a una línea. En este caso podemos simbolizar el arreglo de una dimensión como un segmento de línea. Cada caja se identificará con un número de **índice** distinto. Establezcamos que el primer índice es 1. En el ejemplo se aprecia un arreglo de 4 cajas: caja[1], caja[2], caja[3] y caja[4].



Si pasamos a las dos dimensiones en el espacio nos encontramos con un

plano de ejes x e y, pues bien, podemos simbolizar nuestro array bidimensional como un conjunto de cajas alineadas en columnas y filas. Cada caja se deberá identificar con **dos índices**, en el ejemplo tenemos un arreglo de 3 filas por 4 columnas. Entonces el arreglo estará compuesto por doce variables identificadas desde caja[1,1] a caja[3,4].



Si agregamos una tercera dimensión podemos pensar en un arreglo de 3 dimensiones alto, ancho, profundidad. Y así podemos seguir agregando las dimensiones que querramos.

Declaración de una variable de tipo array

Siempre que se va a utilizar un arreglo se deberá especificar al comienzo cual será la dimensión del mismo. Esto se hace con la palabra reservada Dimensión. En el ejemplo de la variable caja de dos dimensiones la sentencia se escribirá:

```
Dimension caja[3,4]
```

Vectores

Se denominan vectores a los arreglos que poseen una sola dimensión y por lo tanto un sólo número de índice.

Veamos mediante ejemplos cuales son las operaciones más comunes a realizar con un arreglo. Supongamos que tenemos este enunciado:

> *Se requiere un algoritmo que nos muestre de todas las ventas mensuales de un determinado año, cuales meses estuvieron por debajo del promedio de venta mensual.*

Como se resuelve este algoritmo? Primero tenemos que tener todas las ventas mensuales ingresadas para poder promediarlas, son 12 meses en un año entonces ahí encontraremos la cantidad de variables que necesitamos. En algoritmos anteriores habíamos visto que no hace falta recordar todos los valores para obtener el promedio, sin embargo en este caso una vez calculado

el promedio debemos reprocesar la lista de ventas para obtener los que están por debajo del promedio calculado.

Utilizando al metodología Top-Down, pensemos el algoritmo por partes:

1. Carga de los 12 totales de ventas.
2. Cálculo del promedio
3. Búsqueda y muestra de los que están por debajo del promedio.

Vamos a resolverlo entonces.

Primero debemos definir las variables que vamos a utilizar:

```
Dimension ventas[12]
acumulador <- 0
promedio <- 0
```

Carga de un vector

Esta tarea es bastante simple, hay que realizar un ciclo iterativo para cargar los 12 valores. Cada índice en el algoritmo corresponde a un mes, así que aprovecharemos para denominar mes a la variable contadora.

```
Para mes <- 1 Hasta 12 Hacer
    Mostrar "Ingrese las ventas del mes ", mes, ": " Sin Saltar
    Leer ventas[mes]
FinPara
```

Procesamiento del vector cargado

Ahora nos toca revisar los datos cargados acumularlos y contarlos par aobtener el promedio. La verdad que sería innecesario contarlos pues ya sabemos que son 12, pero si habrá que acumularlos para poder obtener el promedio.

```
Para mes <- 1 Hasta 12 Hacer
    acumulador <- acumulador + ventas[mes]
FinPara
promedio <- acumulador / 12
Mostrar "El promedio de venta mensual fue de ", promedio
```

Búsqueda y muestra del vector

Una vez que conocemos el promedio anual, podemos volver a revisar las ventas mensuales para determinar cuales están por debajo del promedio.

```
Mostrar "Meses por debajo del promedio:"
Para mes <- 1 Hasta 12 Hacer
    Si ventas[mes] < promedio Entonces
        Mostrar "El mes ", mes, " -> ", ventas[mes]
    FinSi
FinPara
```

Buenos veamos cómo quedó el programa completo.

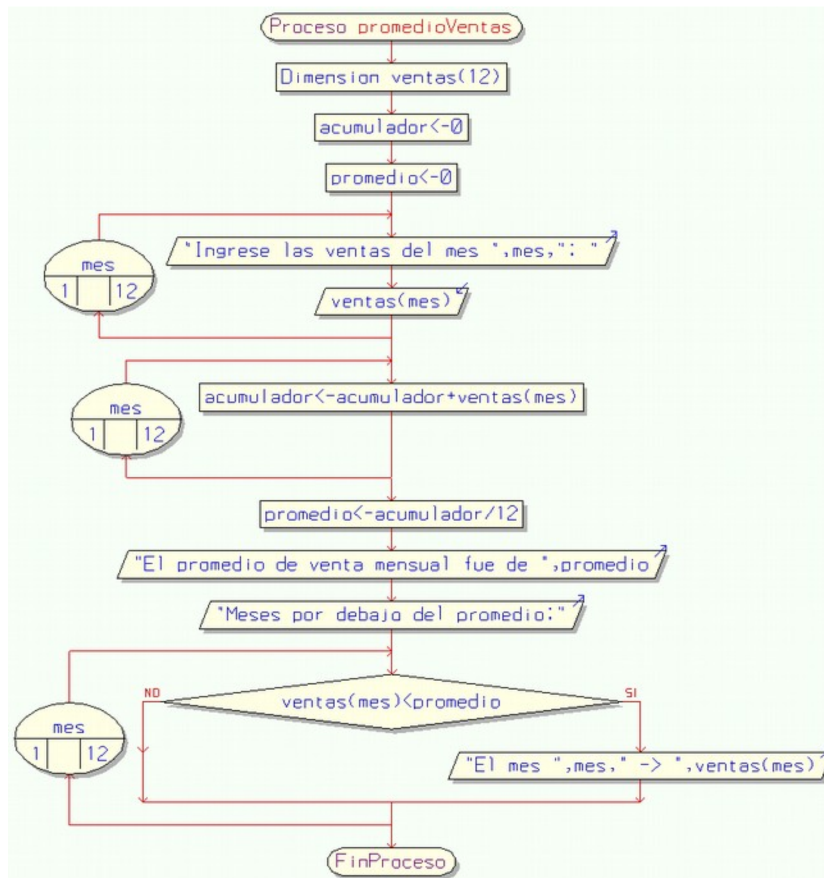
```
Proceso promedioVentas
    Dimension ventas[12]
    acumulador <- 0
    promedio <- 0

    Para mes <- 1 Hasta 12 Hacer
        Mostrar "Ingrese las ventas del mes ", mes, ": " Sin Saltar
        Leer ventas[mes]
    FinPara

    Para mes <- 1 Hasta 12 Hacer
        acumulador <- acumulador + ventas[mes]
    FinPara
    promedio <- acumulador / 12
    Mostrar "El promedio de venta mensual fue de ", promedio

    Mostrar "Meses por debajo del promedio:"
    Para mes <- 1 Hasta 12 Hacer
        Si ventas[mes] < promedio Entonces
            Mostrar "El mes ", mes, " -> ", ventas[mes]
        FinSi
    FinPara
FinProceso
```

y el diagrama de flujo:



Seguramente te habrás dado cuenta que se puede ir acumulando a medida

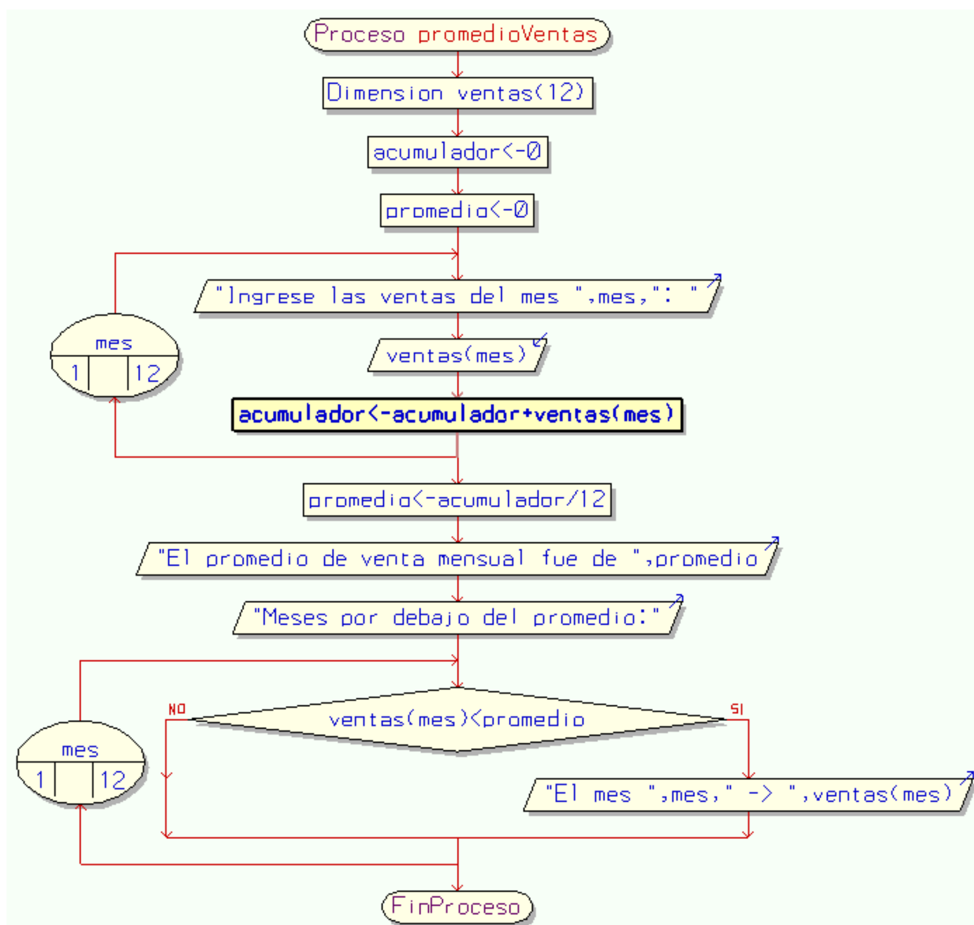
que se va cargando el vector, entonces el algoritmo quedaría un poco más reducido:

```
Proceso promedioVentas
    Dimension ventas[12]
    acumulador <- 0
    promedio <- 0

    Para mes <- 1 Hasta 12 Hacer
        Mostrar "Ingrese las ventas del mes ", mes, ": " Sin Saltar
        Leer ventas[mes]
        acumulador <- acumulador + ventas[mes]
    FinPara
    promedio <- acumulador / 12
    Mostrar "El promedio de venta mensual fue de ", promedio

    Mostrar "Meses por debajo del promedio:"
    Para mes <- 1 Hasta 12 Hacer
        Si ventas[mes] < promedio Entonces
            Mostrar "El mes ", mes, " -> ", ventas[mes]
        FinSi
    FinPara
FinProceso
```

y el diagrama de flujo:



Ordenamiento de un vector

Pensemos en una modificación del enunciado:

> *Se requiere un algoritmo que nos muestre de todas las ventas mensuales de un determinado año, cuales meses estuvieron por debajo del promedio de venta mensual ordenados primero el de menor venta luego el que le sigue y así sucesivamente.*

En otras palabras, queremos saber los que están por debajo del promedio, pero queremos verlos ordenados desde el menor al mayor.

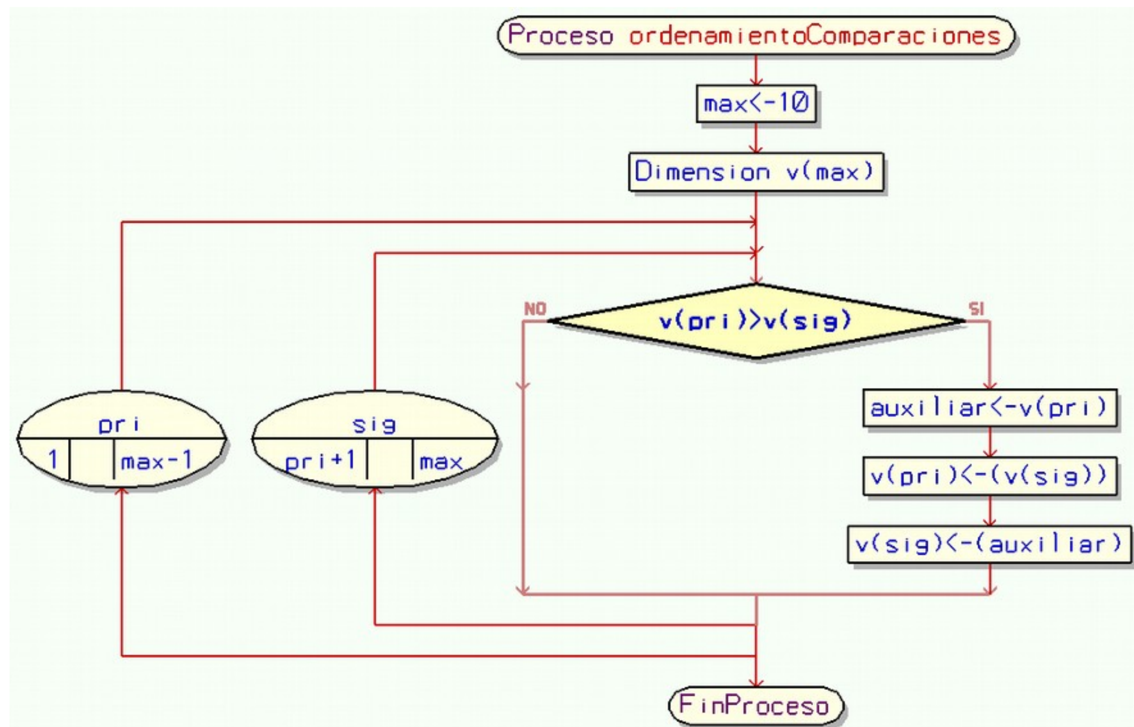
Volvamos a plantear el método top-down:

1. Carga de los 12 totales de ventas y Cálculo del promedio.
2. Ordenamiento del vector.
3. Muestra de las ventas de menor a mayor hasta que encontremos una que supere el promedio.

Vemos que el primer punto se mantiene tal cual, así que tenemos una gran parte hecha, concentrémonos ahora en el ordenamiento... cómo se ordena un vector?

Existen varios algoritmos diseñados para ordenamiento de vectores, cada uno con sus ventajas y desventajas. La idea es poder entenderlos y aplicarlos al algoritmo actual. Vamos a elegir uno simple de entender: el método de ordenamiento de comparaciones sucesivas.

Veamos el diagrama de flujo de este método de ordenamiento:



En este método se utiliza el mecanismo de buscar el menor y ponerlo al principio. Para hacer esto se toma el primer elemento del vector (v[pri]) y se lo va comparando con todos los restantes (v[sig]), cada vez que encontremos uno que sea menor lo intercambiaremos con la primera posición. Al finalizar de

comparar el primer elemento con todos los siguientes estaremos seguros de que el menor quedó en la primera posición. Lo siguiente será repetir la misma búsqueda del menor ahora a partir de la segunda posición y los restantes, luego con la tercera y así hasta la anteúltima, por descarte en la ultima posición quedará el mayor.

```
Proceso ordenamientoComparaciones
    max <- 10
    Dimension v[max]
    Para pri <- 1 Hasta max - 1 Hacer
        Para sig <- pri+1 Hasta max Hacer
            Si v[pri] > v[sig] Entonces
                auxiliar <- v[pri]
                v[pri]     <- v[sig]
                v[sig]     <- auxiliar
            FinSi
        FinPara
    FinPara
FinProceso
```

Este método es bastante simple de entender, pero lamentablemente es de muy baja performance ya que requiere que se comparen los elementos del vector todos con todos por lo tanto al crecer en una unidad el vector se suma un ciclo entero de comparaciones.

Elementos del vector	Cantidad de comparaciones
2	1
3	3 (1 + 2)
4	6 (3 + 3)
5	10 (6 + 4)
6	15 (10 + 5)
7	21 (15 + 6)
8	28 (21 + 7)
9	36 (28 + 8)
10	45 (36 + 9)

Lo invito a prestar atención a esta progresión pues más adelante buscaremos un método para calcularla.

Apliquemos entonces este método de ordenamiento al algoritmo. Pues bien, nos encontramos con un problema, he aquí que el mes de las ventas está asociado al índice así que tendremos que guardar el mes en un vector auxiliar e intercambiarlo a medida que cambiamos las ventas así no perdemos la relación. Vamos a tener que rehacer todo el Algoritmo:

```
Proceso promedioVentas
    Dimension ventas[12]
    Dimension nombreMes[12]
    nombreMes[1] <- "enero"
    nombreMes[2] <- "febrero"
    nombreMes[3] <- "marzo"
    nombreMes[4] <- "abril"
    nombreMes[5] <- "mayo"
    nombreMes[6] <- "junio"
    nombreMes[7] <- "julio"
    nombreMes[8] <- "agosto"
    nombreMes[9] <- "septiembre"
    nombreMes[10] <- "octubre"
    nombreMes[11] <- "noviembre"
    nombreMes[12] <- "diciembre"
    acumulador <- 0
    promedio <- 0

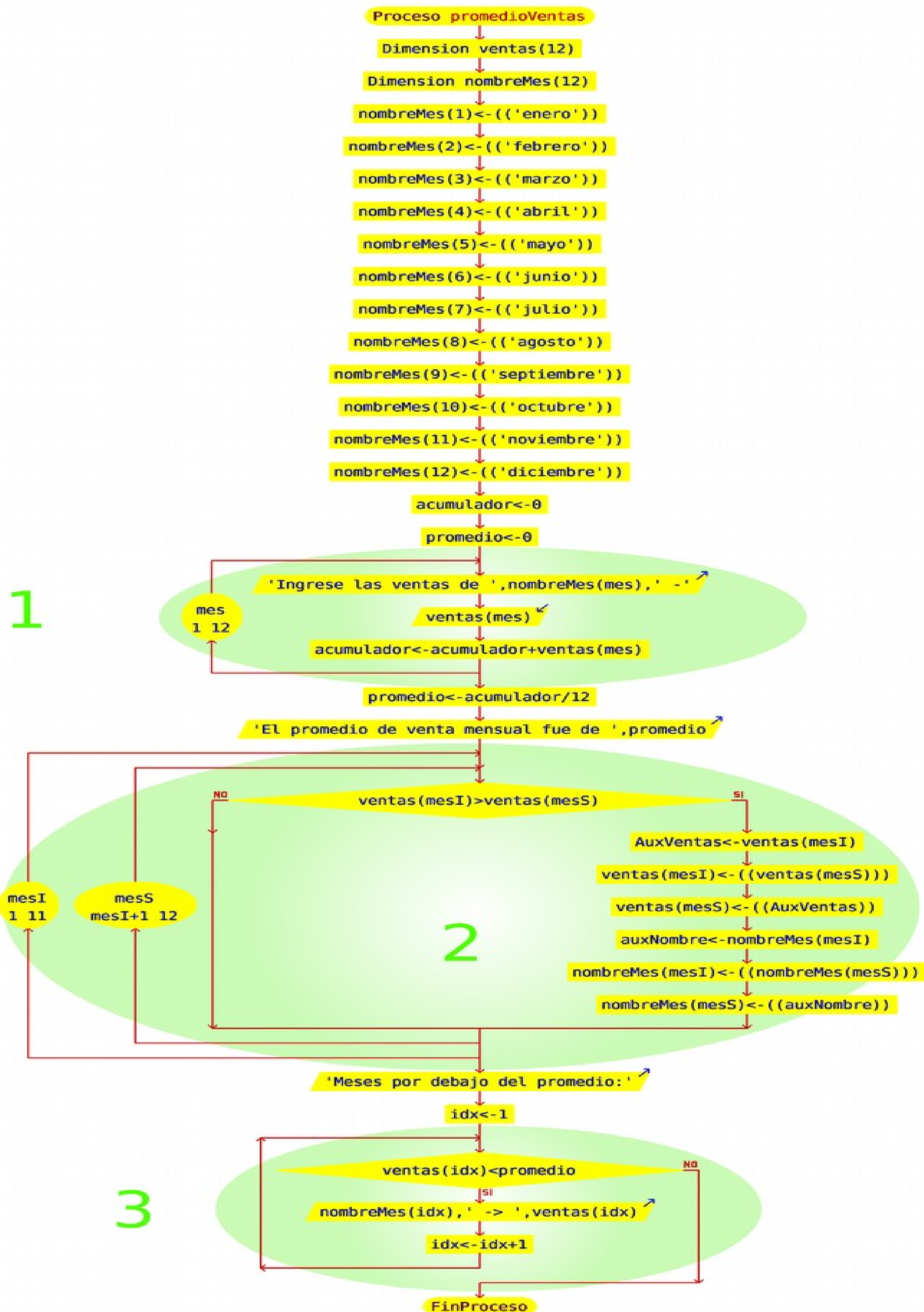
    Para mes <- 1 Hasta 12 Hacer
        Mostrar "Ingrese las ventas de ", nombreMes[mes], " -" Sin Saltar
        Leer ventas[mes]
        acumulador <- acumulador + ventas[mes]
    FinPara
    promedio <- acumulador / 12
    Mostrar "El promedio de venta mensual fue de ", promedio

    Para mesI <- 1 Hasta 11 Hacer
        Para mesS <- mesI + 1 Hasta 12 Hacer
            Si ventas[mesI] > ventas[mesS] Entonces
                AuxVentas     <- ventas[mesI]
                ventas[mesI] <- ventas[mesS]
                ventas[mesS] <- AuxVentas

                auxNombre       <- nombreMes[mesI]
                nombreMes[mesI] <- nombreMes[mesS]
                nombreMes[mesS] <- auxNombre
            FinSi
        FinPara
    FinPara

    Mostrar "Meses por debajo del promedio:"
    idx <- 1
    Mientras ventas[idx] < promedio Hacer
        Mostrar nombreMes[idx], " -> ", ventas[idx]
        idx <- idx + 1
    FinMientras
FinProceso
```

y el diagrama de flujo



Analicemos el diagrama, primero tenemos la declaración de las variables y la definición del vector con los nombres de los meses. A continuación tenemos 3 ciclos importantes, veamoslo uno por uno:

El ciclo 1 corresponde a la carga del vector de ventas y al cálculo del

promedio, tal cual como lo hemos visto anteriormente, el único cambio es en el mensaje: en lugar de mostrar el numero de cada mes, aprovechamos y mostramos su nombre.

El ciclo 2 corresponde al ordenamiento tanto del vector de ventas de mayor a menor como también del vector de nombres (nombreMes) que debe mantener la asociación con el mes de la venta.

El ciclo 3 ahora es una estructura repetitiva condicional, en lugar de una iteración aprovechamos que el vector está ordenado y mientras las ventas sean menores al promedio se mostrarán los datos del vector, apenas superemos el promedio ya no se deberá mostrar nada más.

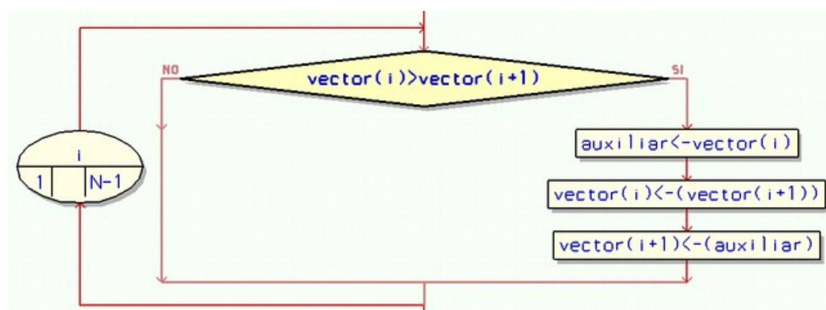
Si usted quiere como tarea adicional podría agregar a continuación otro ciclo para que muestre las ventas ordenadas que superan al promedio, no olvide poner el título “Meses por arriba del promedio” para evitar confusiones en los datos.

Pero, ¿Hay algún método de ordenamiento mejor y que sea facil de entender?

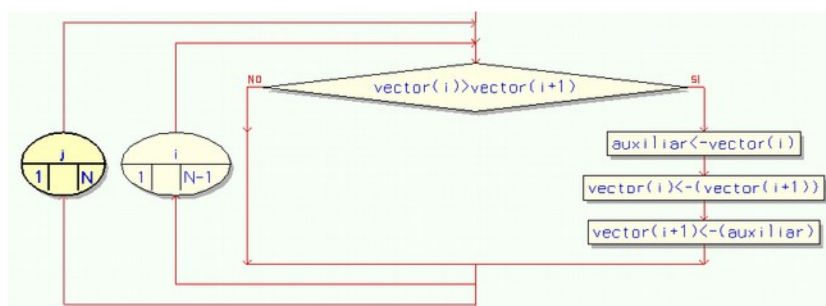
Ordenamiento por Burbujeo

Otro de los métodos de ordenamiento se basa en recorrer todo el vector hasta el anteúltimo elemento y comparar dos elementos adyacentes, si esos se encuentran desordenados entonces se los ordena. En un primer paso podemos observar que al avanzar en las comparaciones siempre el de mayor valor se posiciona inmediatamente al fondo en el primer ciclo, pero los valores menores avanzan una posición por cada ciclo.

Por ejemplo para un vector de tamaño N, un ciclo de ordenamiento sería:



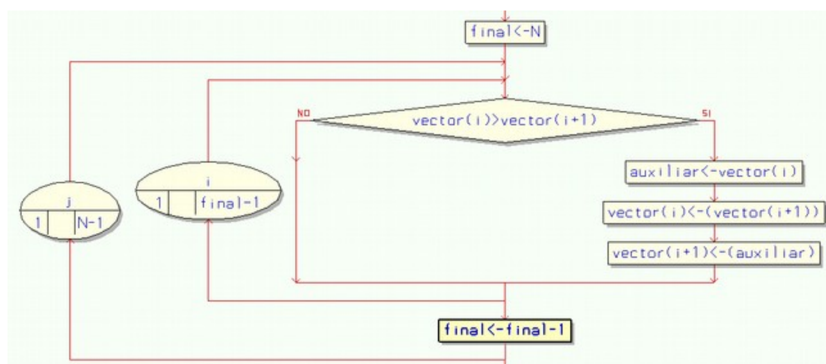
Si no tomamos ninguna consideración para mejorarlo, deberíamos repetir este ciclo tantas veces como elementos tenga el vector menos uno, esto es por que si tenemos la mala suerte de que el menor elemento estuviera al final del vector avanzaría hasta la primera posición de a una posición por ciclo. Entonces el diagrama quedaría así:



La verdad que visto así no presenta ninguna mejora con respecto al método de comparaciones, pero se pueden pensar unas cuantas modificaciones como para que el algoritmo se pueda optimizar.

Primera optimización, acortamiento del ciclo de burbujeo

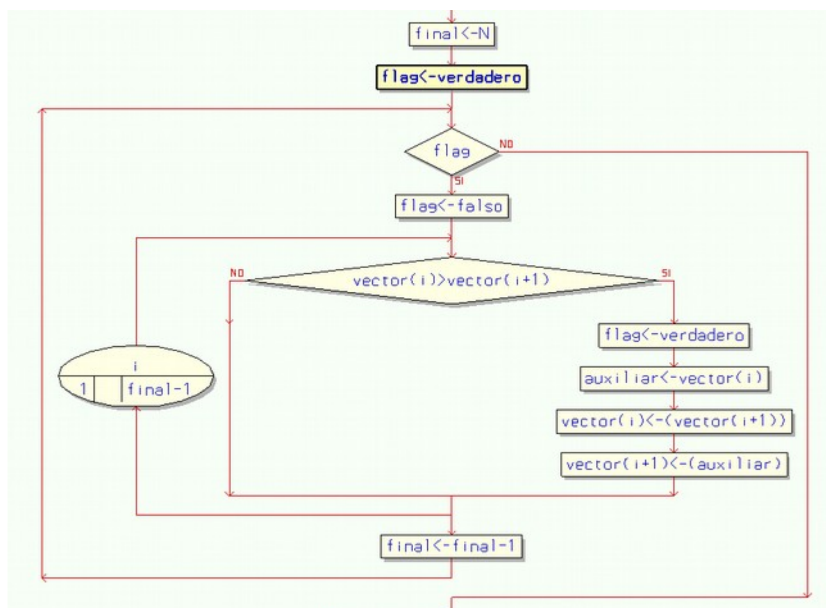
La primera consideración que podemos hacer es precisamente que al efecto de que en cada ciclo se envía el mayor al final, el siguiente ciclo no necesita llegar hasta la última posición. Por lo tanto podemos hacer que el ciclo de comparaciones elimine una comparación por ciclo en forma acumulativa. El diagrama quedaría de la siguiente forma:



Segunda optimización, detectar vector ordenado

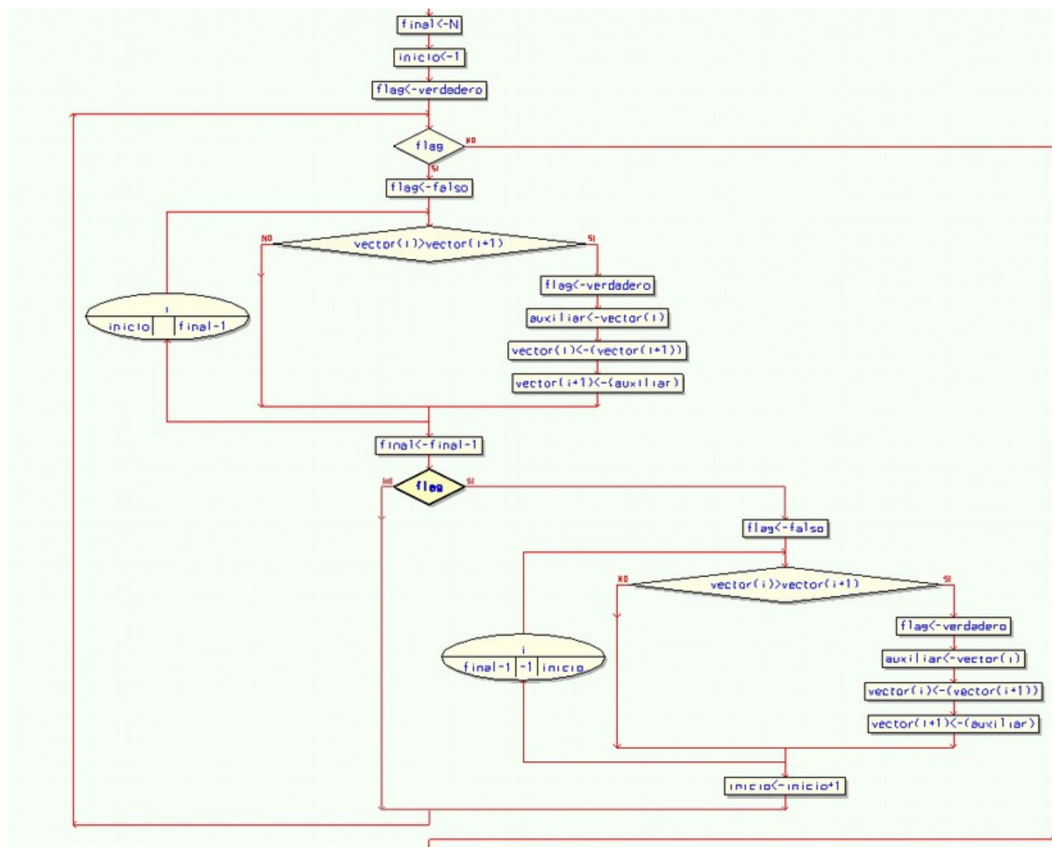
La segunda consideración que podemos hacer es pensar que el algoritmo se plantea como un proceso de fuerza bruta, de forma tal que si el vector se ordenó en un ciclo anterior al final igualmente el proceso se seguirá realizando sin ordenar nada. Para esto podemos encender un flag que indique si hubo un ordenamiento, lo que nos dará una pista de que el vector sigue desordenado. Si el flag está apagado entonces no hace falta seguir ordenando.

Entonces el algoritmo se plantearía de la siguiente forma:



Tercera optimización, ciclos alternados.

Otra optimización que puede hacerse es aprovechar que en un ciclo uno de los extremos se mueve directamente al final y realizar la misma tarea pero en sentido inverso así en el siguiente ciclo tenemos el primer elemento ordenado, esto viene muy bien en el caso de que sean vectores parcialmente ordenados al que se agrega un único elemento desordenado al final. Tal cual como lo hacíamos con el último elemento, debemos ajustar también el extremo inferior en cada ciclo para hacer comparaciones redundantes. Y también contemplaremos el flag en el nuevo ciclo invertido. Ahora el diagrama será un poquito más complejo, pero no tanto.



Parece complejo, pero veamos un ejemplo completo hecho en pseudocódigo:

```
Proceso burbujaOptimizada
    N <- 10   // cantidad de elementos del vector
    Dimension vector[N]
    // cargar el vector
    Para i<- 1 Hasta N Hacer
        vector[i] <- i
    FinPara
    // Desordenar el vector
    Para i<- 1 Hasta N Hacer
        j <- azar(N)+1      // tomar una posición al azar
        auxiliar <- vector[i]
        vector[i] <- vector[j]
        vector[j] <- auxiliar
    FinPara
    // Mostrar el vector desordenado
    mostrar "Vector Desordenado: "
    Para i<- 1 Hasta N Hacer
        mostrar vector[i], " " sin saltar
    FinPara
    mostrar ""
```

```
    final <- N      // variable para acortar el ciclo en el final
    inicio <- 1     // variable para acortar el ciclo en el principio
    flag <- verdadero   // nos indica si hubo un intercambio
    Mientras flag
        flag <- falso  // suponemos que el vector está ordenado
        Para i<- inicio Hasta final-1 Hacer
            Si vector[i] > vector[i+1] Entonces
                flag <- verdadero    // detectamos que el vector estaba desordenado
                auxiliar  <- vector[i]
                vector[i] <- vector[i+1]
                vector[i+1] <- auxiliar
            FinSi
        FinPara
        final <- final - 1  // el último elemento ahora está ordenado con seguridad
        si flag Entonces   // si estaba desordenado
            flag <- falso  // suponemos que el vector ahora está ordenado
            Para i<- final-1 Hasta inicio Con Paso -1 Hacer
                Si vector[i] > vector[i+1] Entonces
                    flag <- verdadero  // detectamos que el vector estaba desordenado
                    auxiliar  <- vector[i]
                    vector[i] <- vector[i+1]
                    vector[i+1] <- auxiliar
                FinSi
            FinPara
            inicio <- inicio + 1  // el primero de los elementos ahora está ordenado
        FinSi
    FinMientras


    Mostrar "Vector Ordenado: "
    Para i<- 1 Hasta N Hacer
        Mostrar vector[i], " " sin saltar
    FinPara
    Mostrar ""

FinProceso
```

Matrices

Archivos

Primero definamos en que consiste un archivo: un archivo es un conjunto de datos (muy a bajo nivel podemos decir que son un conjunto de bits) que se encuentra almacenado en un medio de almacenamiento secundario. Un medio de almacenamiento secundario es accedido a través de un periférico, por lo tanto podemos decir que un archivo se encuentra en una memoria externa.

Generalizando, podemos decir que un archivo es un conjunto de datos que hace referencia a un aspecto de la información, por esto podemos clasificar a los archivos como: archivos de vídeo, archivos de sonido, archivo de imágenes, etc, también como archivos podemos identificar a los archivos fuente (programas en texto hechos por el programador) y archivos ejecutables (programas en binario creados por el programa intérprete), pero especialmente podemos identificar los que más nos interesan: los archivos de datos.

Estructura de un archivo de datos

Hay muchos tipos de archivos de datos, algunos son dependientes de un lenguaje de programación específico o de un programa específico como por ejemplo los archivos DBF que contienen tablas de datos de los lenguajes Dbase, Fox y Clipper, o los archivos MDB que son una base de datos diseñada por el programa ACCESS de Microsoft.

Los archivos de datos que nos interesan son los denominados archivos planos en los cuales se almacena la información agrupadas en registros organizados de forma secuencial, donde cada registro contiene uno o más campos que son datos atómicos equivalentes a los que puede contener una simple variable de memoria.

Si el archivo es de los denominados de tamaño de registro fijo, como los archivos DBF mencionados es bastante facil hacer un esquema de su estructura:

Supongamos que deseamos almacenar datos de un grupo de personas, específicamente de cada persona: el nombre, el Nº de DNI, la edad, el peso y su altura. Entonces podemos representar la estructura el archivo como si se tratara de una tabla bidimensional donde las columnas son los campos y las filas son los registros:

Registro	Nombre	DNI	Edad	Peso	Altura
1	Daniela	34308987	8	43	1.40
2	Oscar	35190831	7	45	1.36
3	Alberto	37073539	5	25	1.08
4	Susana	34254752	9	30	1.45
5	María	33274432	10	32	1.30

Entonces en el archivo se grabará solamente el contenido no grisado de la tabla en formato ASCII, un dato a continuación de otro y un registro inmediatamente después del otro, por lo que si miramos el archivo con un

editor de texto se verá algo como:

```
Daniela,34308987,8,43,1.40←Oscar,35190831,7,45,1.36←
Alberto,37073539,5,25,1.08←Susana,34254752,9,30,1.45
←María,33274432,10,32,1.30←
```

En el ejemplo, cada campo se separa por una coma y cada registro tiene un separador con el símbolo ←. Obsérvese que en esta estructura de archivo no se guardan los nombres de los campos ni los números de registro.

Comandos y funciones relativas a archivos

En el algoritmo se puede hacer relativamente pocas tareas con un archivo, específicamente definiremos 5 tareas:

Abrir un archivo:

La primer tarea para trabajar con archivos en cualquier lenguaje es ejecutar una instrucción que asocie un archivo en un medio de almacenamiento a un canal de comunicación esto se hace con la instrucción:

```
Abrir “nombre de archivo” cómo #1
```

donde *“nombre de archivo”* es el nombre por el cual el sistema operativo reconoce al archivo, puede ser escrito literalmente o también puede ser almacenado como una cadena de texto en una variable.

Por otra parte #1 es el número de canal de comunicaciones abierto, pero en un algoritmo no tendremos por qué limitarnos a procesar un solo archivo, por lo tanto cuando se abra otro archivo llevará el #2, y así sucesivamente.

En algunos lenguajes se suele indicar además si el archivo será abierto para lectura solamente o para escritura para que el sistema operativo sepa como debe manejar los bloqueos (cuando se graba sólo se permite que un único proceso utilice el archivo, por lo tanto se lo bloquea a los demás procesos).

Leer el archivo

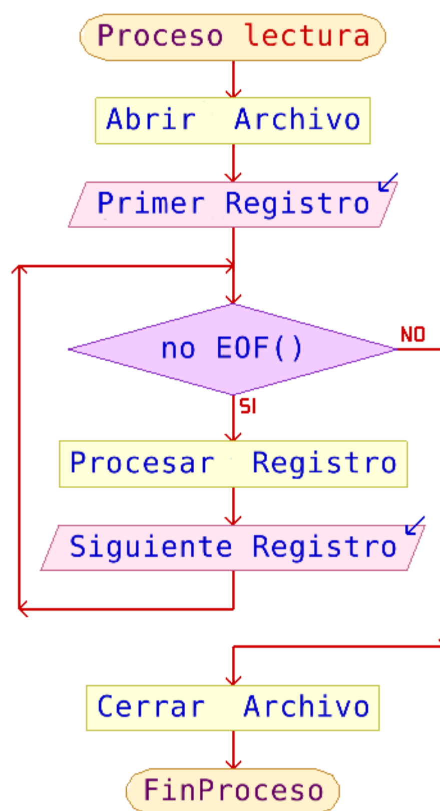
Para obtener datos del archivo a fin de procesarlos por el algoritmo se utilizará la instrucción leer pero agregándole el canal desde donde se obtendrá los datos. La idea es se lea un registro completo, que todos los campos de datos se copien en variables del algoritmo. La instrucción será entonces algo como:

```
Leer #1, variable1, variable2, variable3…
```

Detectar el final del archivo

Pero cuando se lee secuencialmente un archivo siempre hay que procesar un registro tras otro hasta leer todo el archivo o hasta decidir si no se requieren más datos. Cuando se debe procesar el archivo completo, para detectar el final del archivo y no intentar leer datos inexistentes se utiliza una función que devolverá el estado de fin de archivo. La función que se utiliza en la mayoría de los lenguajes (y que utilizaremos en el pseudocódigo) es la sigla EOF() que

corresponde a la palabra inglesa “end of file”, en castellano deberíamos utilizar algo como FDA() (Fin De Archivo) pero por costumbre seguiremos utilizando las sigla en inglés. Más adelante veremos un ejemplo de cómo utilizarla cuando veamos el proceso de lectura secuencial de un archivo.



Un detalle muy importante es que la función EOF() devuelve un valor verdadero cuando se trató de leer más allá del último registro, o sea, cuando la instrucción LEER trajo datos erróneos que no deberán ser procesados. Por lo tanto el uso más acertado de la instrucción de lectura en el ciclo es posterior al proceso de datos.

Esto suena bastante confuso... ¿Se entiende con lo dicho en el párrafo anterior que debo procesar primero el registro y posteriormente leerlo? Es un absurdo, Lo que se debe entender es que al finalizar el proceso del registro en curso se debe leer el próximo registro a procesar, ahí el problema entonces estaría con el primer registro del archivo. Por eso siempre la primera lectura es antes de comenzar el ciclo de proceso de los datos del archivo.

Guardar datos en un archivo

Así como se utiliza la instrucción **leer** para obtener los datos almacenados en un archivo, para guardarlos se utiliza la instrucción **escribir** de la siguiente forma:

```
Escribir #1, variable1, variable2, variable3…
```

Cierre de un archivo

Cuando se finaliza la tarea de proceso asociada a un archivo es buena costumbre indicar que se libere el canal e comunicaciones. Si bien en los sistemas operativos modernos esto se hace automáticamente, es preferible no utilizar más recursos de los que necesitamos por una cuestión de rendimiento y de seguridad en los datos.

Para indicar que el canal de comunicaciones se debe cerrar y por lo tanto el sistema operativo debe tomar los recaudos necesarios para que los datos queden asentados en el archivo (es normal que el sistema operativo utilice una memoria caché asociada al archivo) se debe escribir la instrucción cerrar indicando el canal asociaciado, por ejemplo:

```
Cerrar #1
```

Algoritmos típicos

Creación de un archivo

El proceso necesario para crear un archivo normalmente implica cargar los datos desde el teclado para luego validarlos y si son correctos guardarlos en el archivo. Repitiendo esta tarea hasta que se ingresen todos los datos.

Hagamos un ejemplo, supongamos que queremos tener la lista de nombres, sexo, pesos y alturas de un grupo de personas. Tomaremos de indicación de fin de la carga cuando el nombre sea “X”.

```
Proceso cargaArchivo
 abrir "DATOS.TXT" como #1

 Mostrar "Ingrese el nombre: " Sin Saltar
 Leer nombre
 Mientras mayúsculas(nombre) <> "X" Hacer
       Mostrar "Ingrese sexo (M/F): " Sin Saltar
       Leer sexo
       sexo = mayúsculas(sexo)
       si sexo = "M" o sexo="F" Entonces
             mostrar "Ingrese Peso (en kg): " Sin Saltar
             leer peso
             si peso > 0 y peso < 200 Entonces
                   Mostrar "Ingrese Altura (en m): " Sin Saltar
                   Leer altura
                   si altura > 0.40 y altura < 2.5 Entonces
                         Escribir #1, nombre, sexo, peso, altura
                   Sino
                         Mostrar "Error en altura ingresada"
                   FinSi
             Sino
                   Mostrar "Error en el peso ingresado"
             FinSi
       Sino
             Mostrar "Error en el sexo ingresado"
       FinSi
       Mostrar "Ingrese el nombre: " Sin Saltar
       Leer nombre
 FinMientras

 cerrar #1
FinProceso
```

Lectura del archivo creado.

El proceso de lectura es muy similar en su estructura al anterior, cambian las condiciones del ciclo y el origen de la lectura y la escritura, por lo demás es similar.

Planteamos el algoritmo para leer el archivo generado, pensemos que los datos no hace falta validarlos pues ya están validados en la carga. Entonces el algoritmo resulta mucho más simple:

```
Proceso leerArchivo
 abrir "DATOS.TXT" como #1
```

```
 Mostrar "Datos almacenados: "
 Leer #1, nomb, sex, pe, alt
 Mientras no eof() Hacer
       Mostrar nomb, sex, pe, alt
       Leer #1, nomb, sex, pe, alt
 FinMientras

 cerrar #1
FinProceso#
```

Nota importante: Cuando leemos un archivo debemos utilizar el mismo orden de campos que el algoritmo que se utilizó para su creación. Nótese que se requiere que se respete solamente el orden, no hace falta que los nombres de las variables sean los mismos.

Trabajar con 2 archivos

Veamos un ejemplo donde utilicemos dos archivos. Supongamos que tenemos el archivo con la clasificación del índice de masa corporal (ver en Wikipedia IMC, https://es.wikipedia.org/wiki/%C3%8Dndice_de_masa_corporal).

Supongamos que el archivo se llama “IMC.DAT” y contiene el estado y el rango de valores (desde y hasta) que abarca ese estado, con los siguientes valores

```
Bajo peso                         0            18.5
Normal                         18.5              25
Sobrepeso                        25              30
Obesidad leve                    30              35
Obesidad media                   35              40
Obesidad mórbida                 40             100
```

la forma de calcular el IMC es el peso dividido la altura al cuadrado (o sea IMC = kg/m^2).

Pues bien, la idea sería mostrar nuevamente el archivo pero ahora indicando según su IMC en que estado nutricional se encuentra la persona.

El algoritmo entonces necesitará procesar dos archivos, el archivo IMC y el cargado con pesos y alturas. Para agilizar el proceso y no tener que andar buscando el rango de IMC dentro del archivo (abriendo el archivo cada vez que se busca un rango) y como los datos del IMC son realmente pocos, entonces cargaremos dos arreglos: un vector y una matriz, luego obtendremos la clasificación de los datos cargados en memoria.

```
Proceso muestraIMC
 dimension IMC_estado[6]
 dimension IMC_valor[6,2]
 i <- 1

 // primer paso, cargar el archivo IMC dentro de los arreglos
 abrir "IMC.DAT" como #1
 Leer #1, IMC_estado[i], IMC_valor[i,1], IMC_valor[i,2]
 Mientras no eof(#1) Hacer
       i <- i + 1
```

```
        Leer #1, IMC_estado[i], IMC_valor[i,1], IMC_valor[i,2]
  FinMientras
  cerrar #1

  // Ahora procesar el archivo con los pesos y alturas
  abrir "DATOS.TXT" como #2
  Mostrar "Estado nutricional: "
  Leer #2, nombre, sexo, peso, altura
  Mientras no eof(#2) Hacer
        IMC <- peso / ( altura ^ 2 )
        Mostrar nombre, "Estado nutricional: " Sin Saltar
        Mostrar estado_IMC(IMC, IMC_Estado, IMC_valor )
        Leer #2, nombre, sexo, peso, altura
  FinMientras

  cerrar #2
FinProceso

Funcion estado <- estado_IMC( IMC, e, v)
  i <- 1
  // Buscando el rango que corresponde al IMC
  // mientras no supere el final del arreglo i <= 6
  // y no encuentre que el IMC entra dentro de un rango
  Mientras i <= 6 y no ( IMC > v[i,1] y IMC <= v[i,2] ) Hacer
        i <- i + 1
  FinMientras
  si i = 7 Entonces
        estado <- "Fuera de rango!"
  Sino
        estado <- e[i]
  FinSi
FinFuncion
```

Corte de control

Se denomina corte de control a un algoritmo típico en el proceso de datos que consiste en procesar un archivo que obligatoriamente debe estar ordenado por grupos de varios niveles para obtener totales por grupo.

Veamos un ejemplo: supongamos que tenemos un archivo que contiene una lista de personal de una empresa que tiene varias sucursales en todo el país, y que esta lista tiene los siguientes campos:

- *Código de provincia*
- *Nombre de sucursal*
- *Cargo (1-encargado de sucursal, 2- administrativo, 3-operativo)*
- *Apellido y Nombres del empleado*
- *DNI del empleado*

Supongamos que el archivo está ordenado primero por código de provincia, luego por Nombre de sucursal y por último está ordenado por cargo. Además tenemos otro archivo cuyos registros contienen dos campos: código de provincia y nombre de provincia.

Queremos realizar un algoritmo que nos permita conocer cuantos empleados de cada tipo de cargo hay en cada sucursal. Cuantos empleados en total hay en

cada provincia y cuantos empleados hay en total en todo el país. De paso, para complicar el algoritmo queremos saber cuantos empleados tiene la sucursal que más empleados tiene.

Digamos que por pantalla queremos que nos salga la información más o menos así:

```
Listado de Cantidad de personal
Provincia: XXXXXXXXXXXXXX
  Sucursal: XXXXXXXXXXXXXXXXXXXXX
    Encargados: 99
    Administrativos: 99
    Operativos: 99
  Sucursal: XXXXXXXXXXXXXXXXXXXXX
    Encargados: 99
    Administrativos: 99
    Operativos: 99
  Total empleados provincia: 9999
Provincia: XXXXXXXXXXXXXX
  Sucursal: XXXXXXXXXXXXXXXXXXXXX
    Encargados: 99
    Administrativos: 99
    Operativos: 99
  Sucursal: XXXXXXXXXXXXXXXXXXXXX
    Encargados: 99
    Administrativos: 99
    Operativos: 99
  Total empleados provincia: 9999
Total empleados del país: 9999999
La sucursal con más empleados tiene: 999 empleados.
```

Un posible algoritmo para solucionar el problemas sería:

```
Proceso CorteControl
    empleadosTotal <- 0
    maxEmpleadosSuc <- 0
    Dimension Provincias[24,2]
    cargaProvincias(Provincias)

    Abrir "personal.dat" Como #2
    // Leo el primer registro
    Leer #2, codProv, NombSuc, Cargo, AyN, DNI
    // Titulos generales
    Mostrar "Listado de Cantidad de Personal"
    // Corte de nivel general
    Mientras no eof(#1) Hacer
        // Titulos de 1er corte (Provincia)
        Mostrar "  Provincia: ", nombreProvincia( codProv, Provincias )
        // Variable auxiliar para detectar el corte de control
        auxCodProv <- codProv
        // Acumulador por provincia a 0
        empleadosProv <- 0
        // Primer Corte de control
```

```
        Mientras no eof(#1) y auxCodProv = codProv Hacer
            // Titulos de 2do corte (Sucursal)
            Mostrar "    Sucursal: ", NombSuc
            // Variable auxiliar para detectar el corte de control
            auxNombSuc <- NombSuc
            // Acumulador por sucursal a 0
            empleadosSuc <- 0
            // Segundo Corte de control
            Mientras no eof(#1) y auxCodProv = codProv y auxNombSuc = NombSuc Hacer
                // Variable auxiliar para detectar el corte de control
                auxCargo <- Cargo
                // Acumulador por cargo a 0
                empleadosCargo <- 0
                // Tercer Corte de control
                Mientras no eof(#1) y auxCodProv = codProv y auxNombSuc = NombSuc
                                                     y auxCargo = Cargo Hacer
                    // Cuento los empleados
                    empleadosCargo <- empleadosCargo + 1
                    // Leo el siguiente registro
                    Leer #2, codProv, NombSuc, Cargo, AyN, DNI
                FinMientras
                // Total del 3er Corte de control
                Mostrar "    ", nombreCargo( auxCargo ), empleadosCargo
                // Acumulo para el nivel anterior
                empleadosSuc <- empleadosSuc + empleadosCargo
            FinMientras
            // Total del 2do Corte de control
            // Busco el máximo de empleados por sucursal
            Si maxEmpleadosSuc < empleadosSuc Entonces
                maxEmpleadosSuc <- empleadosSuc
            FinSi
            // Acumulo para el nivel anterior
            empleadosProv <- empleadosProv + empleadosSuc
        FinMientras

        // Total del 1er Corte de control
        Mostrar "  Total empleados Provincia: " , empleadosProv
        // Acumulo para el nivel anterior
        empleadosTotal <- empleadosTotal + empleadosProv
    FinMientras
    // Fin del proceso – Totales Generales
    Cerrar #2
    Mostrar "Total empleados del País: " , empleadosTotal
    Mostrar "La sucursal con más empleados tiene: ", maxEmpleadosSuc, " empleados"
FinProceso

// con este subproceso Cargo el archivo de provincias al vector
SubProceso cargaProvincias( Por Referencia Prov )
    i <- 0
    Abrir "provincias.dat" Como #1
    Leer #1, codProv, nombProv
    Mientras no eof(#1) Hacer
        i <- i + 1
        Prov[i,1] <- codProv
        Prov[i,2] <- nombProv
        Leer #1, codProv, nombProv
    FinMientras
    Cerrar #1
FinSubProceso

// Busco el nombre de la provincia por su código
Funcion nombre <- nombreProvincia( codigo, Provincias )
    i <- 1
```

```
    // En el siguiente ciclo busco coincidencia en el código
    // y verifico no pasarme del final del vector
    Mientras i <= 24 y codigo <> Provincias[i,1] Hacer
        i <- i + 1
    FinMientras
    // Al finalizar el ciclo analizo porqué salió del ciclo
    si i = 25 Entonces
        // si terminó porque se pasó
        nombre <- "Provincia Desconocida"
    Sino
        // sino terminó porque encontró el código
        nombre <- Provincias[i,2]
    FinSi
FinFuncion

// Devuelvo el nombre del cargo por su código
Funcion nombre <- nombreCargo( codCargo )
    Segun codCargo Hacer
        1:
            nombre <- "Encargados: "
        2:
            nombre <- "Administrativos: "
        3:
            nombre <- "Operativos: "
        De Otro Modo:
            nombre <- "Código erróneo: "
        Fin Segun
FinFuncion
```

Si prestamos atención al algoritmo, hay dos campos que no han sido utilizados por el algoritmo, Apellido y Nombre y DNI del empleado. Sin embargo como estos datos están físicamente guardados en el archivo y ocupan lugar, deben ser leídos obligatoriamente para que el sistema controle el final del registro y no se pierda la sincronización de datos.